Spring Data JPA

https://spring.io/projects/spring-data
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#preface

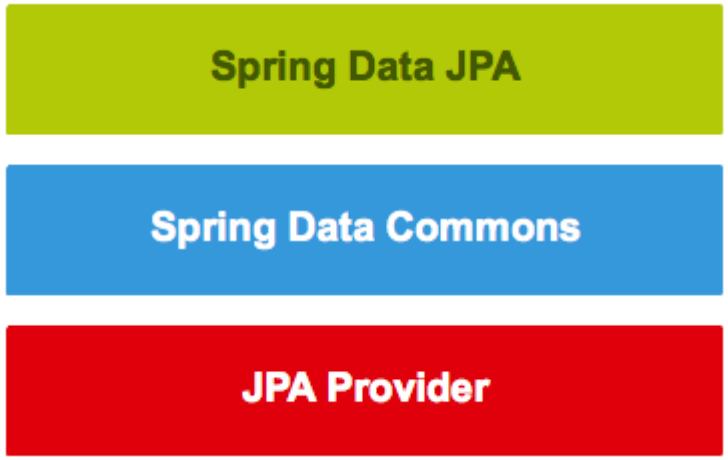
- dasturlash.uz

JPA

- JPA or Java Persistence API is the Java specification for accessing, managing and persisting data between Java classes or objects and relational database.
- JPA is not an implementation or product, it is just a specification.
- It contains set of interfaces which need to be implemented.
- It is a framework that provides an extra layer of abstraction on the JPA implementation.
- The repository layer will contain three layers as mentioned below.

Spring Data

- **What Spring Data JPA Is?**
- **Spring Data JPA is not a JPA provider**
- It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider.
- If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:



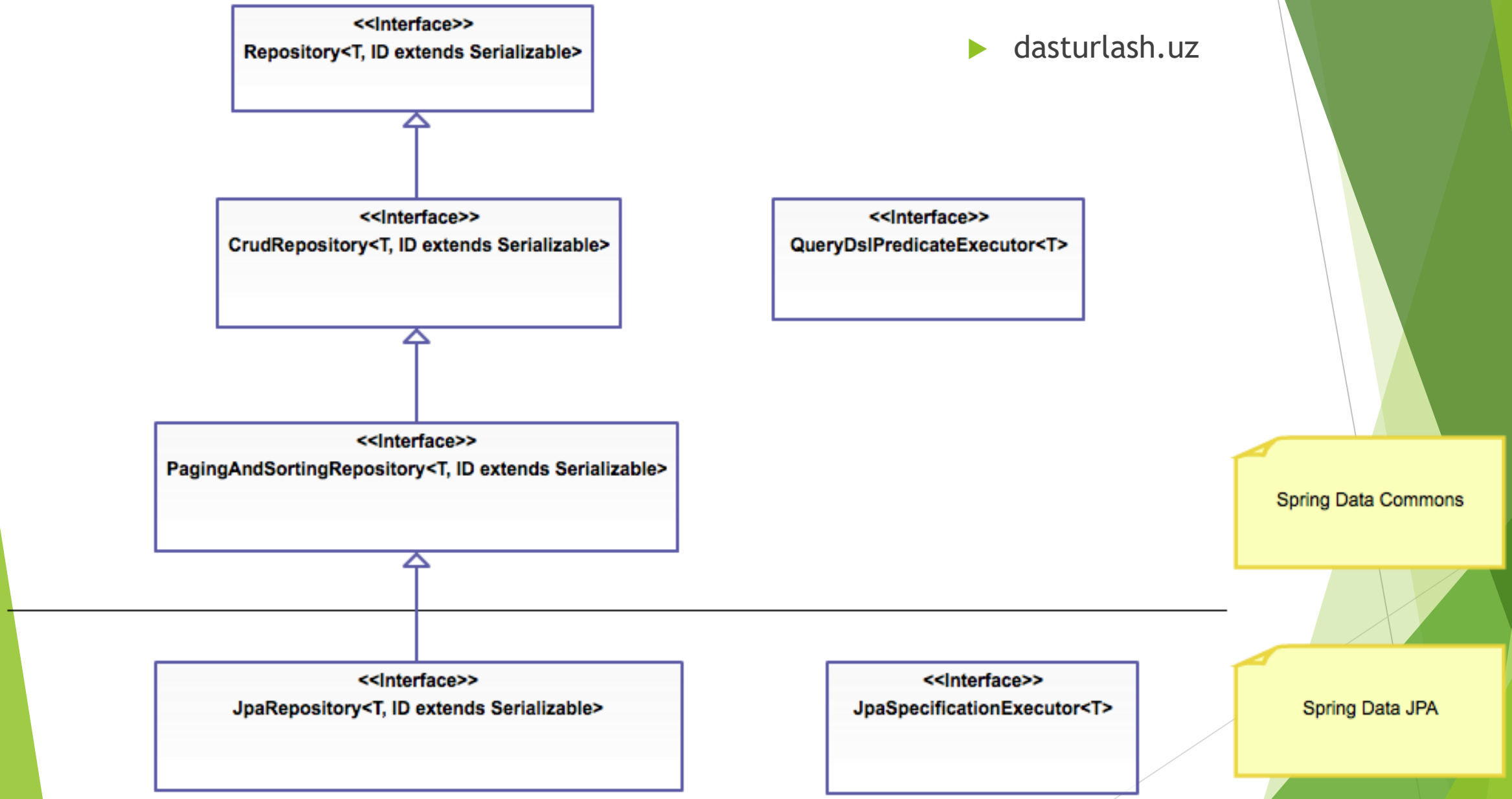
- dasturlash.uz

Spring Data

- **Spring Data JPA**: – This provides spring data repository interfaces which are implemented to create JPA repositories.
- **Spring Data Commons**: – It provides the infrastructure that is shared between data store specific spring data projects.
- **The JPA** provider which implements the JPA persistence API.

- https://www.amitph.com/jpa-and-spring-data-jpa/

- dasturlash.uz



Spring Data

- Spring Data Commons project provides the following interfaces:

- Repository<T, ID extends Serializable> interface
- CrudRepository<T, ID extends Serializable> interface
- PagingAndSortingRepository<T, ID extends Serializable> interface
- QueryDslPredicateExecutor interface

Repository<T,ID>

- This interface is a marker interface
  - It captures the type of the managed entity and the type of the entity's id.
  - It helps the Spring container to discover the "concrete" repository interfaces when classpath is scanned.

- Type Parameters:
- T - the domain type the repository manages
- ID - the type of the id of the entity the repository manages

```
@Indexed
public interface Repository<T, ID> {

}
```

- dasturlash.uz

CrudRepository<T, ID extends Serializable>

- It provides CRUD operations for the managed entity.

```
@NoRepositoryBean
public interface CrudRepository < T, ID > extends Repository < T, ID > {

 <S extends T > S save(S entity);

 <S extends T > Iterable < S > saveAll(Iterable < S > entities);

 Optional < T > findById(ID id);

 boolean existsById(ID id);

 Iterable < T > findAll();
 terable < T > findAllById(Iterable < ID > ids);
 long count();
```

-

```
void deleteById(ID id);
 void delete(T entity);
 void deleteAll();

}
```

- dasturlash.uz

The PagingAndSortingRepository<T, ID extends Serializable>

- It is an extension of *CrudRepository* to provide additional methods to retrieve entities using the pagination and sorting abstraction.

  @NoRepositoryBean

- public interface PagingAndSortingRepository < T, ID > extends CrudRepository < T, ID > {

  Iterable < T > findAll(Sort sort);

  Page < T > findAll(Pageable pageable);

- }

QueryDslPredicateExecutor

- It interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using *QueryDsl* Predicate objects

```
public interface QuerydslPredicateExecutor < T > {


  Optional < T > findOne(Predicate predicate);
   Iterable < T > findAll(Predicate predicate);
   Iterable < T > findAll(Predicate predicate, Sort sort);
   Iterable < T > findAll(Predicate predicate, OrderSpecifier << ? > ...orders);
   Iterable < T > findAll(OrderSpecifier << ? > ...orders);
   Page < T > findAll(Predicate predicate, Pageable pageable);
   long count(Predicate predicate);
   boolean exists(Predicate predicate);


}
```

- dasturlash.uz

Spring data

- **Spring Data JPA** project provides the following interfaces:
- JpaRepository<T, ID extends Serializable> interface
- JpaSpecificationExecutor interface

JpaRepository<T, ID extends Serializable>

- It is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.
- public interface JpaRepository < T, ID > extends PagingAndSortingRepository < T, ID > , QueryByExampleExecutor < T > {

  }

JpaSpecificationExecutor interface

- It interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using Specification objects that use the JPA criteria API.

```
public interface JpaSpecificationExecutor<T> {

Optional<T> findOne(@Nullable Specification<T> spec);
List<T> findAll(@Nullable Specification<T> spec);

Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);

List<T> findAll(@Nullable Specification<T> spec, Sort sort);

long count(@Nullable Specification<T> spec);

}
```

Spring Data JPA - Query Creation from Method Names

- dasturlash.uz

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1

Spring Data JPA - Query Creation from Method Names

- dasturlash.uz

Keyword	Sample	JPQL snippet
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

Spring Data JPA - Query Creation from Method Names

- https://docs.spring.io/spring-data/jpa/docs/1.3.0.RELEASE/reference/html/jpa.repositories.html

Different method names

- The first part – like *find* – is the *introducer* and the rest – like *ByName* – is the *criteria*.

- *Introducer : find, query, get, read, count, delete*
- *Criteria: By…*, AllBy, FirstBy, TopBy, DistinctBy, DistinctFirstBy, DistinctTopBy

findBy	queryBy	readBy	getBy
findAllBy	queryAllBy	readAllBy	getAllBy
findDistinctBy	queryDistinctBy	readDistinctBy	getDistinctBy
findDistinctFirstBy	queryDistinctFirstBy	readDistinctFirstBy	getDistinctFirstBy
findDistinctTopBy	queryDistinctTopBy	readDistinctTopBy	getDistinctTopBy
findFirstBy	queryFirstBy	readFirstBy	getFirstBy
findTopBy	queryTopBy	readTopBy	getTopBy
countBy	deleteBy	findTop3By	
countAllBy	deleteAllBy		
countDistinctBy			

Pageabal

- dasturlash.uz

Pageabal

```
public interface Page<T> extends Slice<T> {

  static <T> Page<T> empty();
  static <T> Page<T> empty(Pageable pageable);
  long getTotalElements();
  int getTotalPages();
  <U> Page<U> map(Function<? super T,? extends U> converter);
}


public interface Slice<T> extends Streamable<T> {
  int getNumber();
  int getSize();
  int getNumberOfElements();
  List<T> getContent();
  boolean hasContent();
  Sort getSort();
  boolean isFirst();
  boolean isLast();
  boolean hasNext();
  boolean hasPrevious();
  ...
}
```

- dasturlash.uz

Page and Pageabal

- Pageable paging = PageRequest.of(page, size);

- page: zero-based page index, must NOT be negative.
- size: number of items in a page to be returned, must be greater than 0.

Pageabal – Example 1

```
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort var1);

  Page<T> findAll(Pageable var1);

}
```

```
Pageable pageable = PageRequest.of(0, 30);

Page<ProfileEntity> page = this.profileRepository.findAll(pageable);

int totalPage = page.getTotalPages(); // for number of total pages.
long totalElements = page.getTotalElements(); //  for total items stored in database.
int currentPage = page.getNumber(); // for current Page.
List<ProfileEntity> profileList = page.getContent(); // to retrieve the List of items in the page.
```

Pageabal – Example 2

```
Pageable pageable = PageRequest.of(0, 30);

List<ProfileEntity> list = this.profileRepository.findAllByName("Vali", pageable);

list.forEach(profileEntity -> System.out.println(profileEntity));
```

Sort

- dasturlash.uz

Sort

- 1. By using methodNameQuery

```
List<Passenger> findByOrderByNameAsc();
List<Passenger> findByOrderByNameDesc();

List<Passenger> findByNameOrderByNameAsc();
```

- **2. Sorting with a *Sort* Parameter**

```
Sort sort = Sort.by(Sort.Direction.ASC, "name");
List<ProfileEntity> pList= profileRepository.findAll(sort);
```

Sort.by () --- metodni turli ko’rinishlari bor. Ularni ham ishlatsa bo’ladi

- dasturlash.uz

Sort

- 3. **Sorting with a *Sort and MethodName query***

```
Sort sort = Sort.by(Sort.Direction.ASC, "name");
List<ProfileEntity> page = profileRepository.findByName("Ali", sort);
```

- 4. **Sort with Pageable Object**

```
Sort sort = Sort.by(Sort.Direction.ASC, "name");
Pageable pageable = PageRequest.of(0, 1, sort);
Page<ProfileEntity> page  = profileRepository.findAll(pageable);
```

```
Page<ProfileEntity> page = profileRepository.findByName("Ali", pageable);
```

JPQL and HQL

- dasturlash.uz

JPQL - 1

- **By default, the query definition uses JPQL.**
- The JPQL (Java Persistence Query Language) is an object-oriented query language which is used to perform database operations on persistent entities.
- Instead of database table, JPQL uses entity object model to operate the SQL queries.
- Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.

JPQL - 2

- JPQL is an extension of Entity JavaBeans Query Language (EJBQL), adding the following important features to it: -
  - It can perform join operations.
  - It can update and delete data in a bulk.
  - It can perform aggregate function with sorting and grouping clauses.
  - Single and multiple value result types.

- JPQL Features
  - It is a platform-independent query language.
  - It is simple and robust.
  - It can be used with any type of database such as MySQL, Oracle.
  - JPQL queries can be declared statically into metadata or can also be dynamically built in code.

Creating Queries in JPQL

- JPQL provides two methods that can be used to access database records. These methods are: -
  - Query createQuery(String name) - The createQuery() method of EntityManager interface is used to create an instance of Query interface for executing JPQL statement.
- Query query = em.createQuery("Select s from StudentEntity s");

- This method creates dynamic queries that can be defined within business logic.
  - Query createNamedQuery(String name) - The createNamedQuery() method of EntityManager interface is used to create an instance of Query interface for executing named queries.
- @NamedQuery(name = "find name" , query = "Select s from StudentEntity s")

JPQL

- In Spring Data JPQL can be done using @Query annotation.

@Query

- dasturlash.uz

@Query

- Spring Data provides many ways to define a query that we can execute. One of these is the *@Query* annotation.
- In order to execute SQL query in Spring Data repository method, we can **annotate the method with the *@Query* annotation — its *value* attribute contains the JPQL or SQL to execute.**

@Query - Indexed Query Parameters

- There are two possible ways that we can pass method parameters to our query: indexed and named parameters.

- **@Query("select u from User u where u.emailAddress = ?1“)**
  User findByEmailAddress(String emailAddress);
- @Query("select u from User u where u.firstname like %?1“)
  List<User> findByFirstnameEndsWith(String firstname);
- @Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2") User **findUserByStatusAndName**(Integer status, String name);

@Query - Named Parameters

- We can also **pass method parameters to the query using named parameters.** We define these using the *@Param* annotation inside our repository method declaration.
- @Query("SELECT t.title FROM Todo t where t.id = :id“)
  Optional<String> findTitleById(@Param("id") Long id)

- @Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")
  User **findUserByUserStatusAndUserName**(@Param("status") Integer userStatus, @Param("name") String userName);

@Query - Collection Parameter

- Let's consider the case when the *where* clause of our JPQL or SQL query contains the *IN* (or *NOT IN*) keyword:
- @Query(value = "SELECT u FROM User u WHERE u.name IN :names") List<User> **findUserByNameList**(@Param("names") Collection<String> names);

@Query - NativeQuery

- **Creating SQL Queries**
- The @Query annotation allows for running native queries by setting the nativeQuery flag to true.

```
@Query(value = "select * from users where first_name like %?1“ , nativeQuery = true)
List<User> findByFirstnameEndsWith(String firstname);
```

```
@Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1“ , nativeQuery = true)
User findByEmailAddress(String emailAddress);
```

@Query –Pagination 1

- Pagination allows us to return just a subset of a whole result in a *Page*. This is useful, for example, when navigating through several pages of data on a web page.
- Another advantage of pagination is that the amount of data sent from server to client is minimized. By sending smaller pieces of data, we can generally see an improvement in performance.
- **JPQL**
  - @Query(value = "SELECT u FROM User u ORDER BY id")
    Page<User> **findAllUsersWithPagination**(Pageable pageable);
- **Native**
  - @Query(value = "SELECT * FROM Users ORDER BY id", countQuery = "SELECT count(*) FROM Users", nativeQuery = true) Page<User> **findAllUsersWithPagination**(Pageable pageable);

- dasturlash.uz

@Query – Pagination 2

- **When we use JPQL for a query definition, then Spring Data can handle sorting without any problem**

- ```
  @Query(value = "SELECT * FROM profile WHERE name = ?1",
        countQuery = "SELECT count(*) FROM profile WHERE name = ?1",
        nativeQuery = true)
  Page<ProfileEntity> findByLastname(String lastname, Pageable pageable);
  ```

- ```
  Pageable pageable = PageRequest.of(0, 15);
  Page<ProfileEntity> page = profileRepository.findByLastname("Vali", pageable);

  int totalPage = page.getTotalPages();
  long totalElements = page.getTotalElements();
  int currentPage = page.getNumber();
  List<ProfileEntity> profileList = page.getContent();
  ```

- dasturlash.uz

@Modifying

- dasturlash.uz

@Modifying

- **The *@Modifying* annotation is used to enhance the *@Query* annotation to execute not only *SELECT* queries but also *INSERT*, *UPDATE*, *DELETE*, and even *DDL* queries.**

- ```
  @Modifying
  @Query("update Profile p set p.firstname = ?1 where p.lastname = ?2")
  int setFixedFirstnameFor(String firstname, String lastname);
  ```

- ```
  @Modifying
  @Query("update User u set u.active = false where u.lastLoginDate < :date")
   void deactivateUsersNotLoggedInSince(@Param("date") LocalDate date);
  ```

- dasturlash.uz

@Modifying

- Delete Query . As we can see, this method returns an integer.
- **It's a feature of Spring Data JPA *@Modifying* queries that provides us with the number of updated entities.**
- @Modifying
  @Query("delete User u where u.active = false")
  int deleteDeactivatedUsers();

- *DDL* query - let's add a *deleted* column to our *USERS* table with a *DDL* query:
- @Modifying
  @Query(value = "alter table USERS.USERS add column deleted int(1) not null default 0",
  nativeQuery = true)
  void addDeletedColumn();

- dasturlash.uz

@Modifying

- If we do not user @Modifying annotation

- @Query("delete User u where u.active = false") **int deleteDeactivatedUsersWithNoModifyingAnnotation**();

- After execute the above method, we get an *InvalidDataAccessApiUsage* exception:

@Modifying

- **If our modifying query changes entities contained in the persistence context, then this context becomes outdated.**
- One way to manage this situation is to clear the persistence context.
- By doing that, we make sure that the persistence context will fetch the entities from the database next time.
- However, we don't have to explicitly call the *clear()* method on the *EntityManager*.
- We can just use the *clearAutomatically* property from the *@Modifying* annotation:
- @Modifying(clearAutomatically = true)
- **That way, we make sure that the persistence context is cleared after our query execution.**
- But, what if our persistence context contained unflushed changes? Therefore, clearing it would mean dropping unsaved changes. Fortunately, there's another property of the annotation we can use – *flushAutomatically*:
- @Modifying(flushAutomatically = true)

@NamedQuery

- dasturlash.uz

@NamedQuery and *@NamedQueries*

- These annotations let you define the query in native SQL by losing the database platform independence.

```
@Entity

@NamedQuery (name = "Author.findByFirstName", query = "FROM Author
                    WHERE firstName = ?1")

@NamedQuery (name = "Author.findByFirstNameAndLastName", query =
  "SELECT a FROM Author a WHERE a.firstName = ?1 AND a.lastName = ?2")

public class Author { ... }
```

@NamedQuery and *@NamedQueries*

- In @Entity class

```
@NamedNativeQuery(name = "Author.findByFirstName", query = "SELECT *
FROM author WHERE first_name = ?", resultClass = Author.class)

@NamedNativeQuery(name = "Author.findByFirstNameAndLastName", query =
"SELECT * FROM author WHERE first_name = ? AND last_name = ?", resultClass
= Author.class)

public class Author { ... }
```

@NamedQuery and *@NamedQueries*

- @NamedNativeQuery(name = "User.findByEmailAddress", query = "select * from users where email_address = ?1", resultClass = User.class)

```
@NamedNativeQueries(
 value = { @NamedNativeQuery(
                    name = "User.findByLastname", query = "select
                                                * from users where lastname = ?1",
  resultClass = User.class) })

public class User {…..}
```

@NamedQuery and *@NamedQueries Using*

- Just use defined name in

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
   List<Author> findByFirstName(String firstName);

   List<Author> findByFirstNameAndLastName(String firstName, String lastName);

}
```

JOIN

- dasturlash.uz

Spring Data JOIN – example tables

```
@Entity
@Table(name = "student")
public class StudentEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private Integer age;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id")
    private AddressEntity address;
}
```

```
@Getter
@Setter
@Entity
@Table(name = "address")
public class AddressEntity {
    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Integer id;
    @Column
    private String name;
}
```

JOIN – example 1

- Get Student by Address Id

JOIN – example 1 solution

- Get Student without join In Method name

```
public interface StudentRepository ... {
    StudentEntity findByAddress(AddressEntity addressEntity);
}
```

```
   public void getStudentByAddressId(Integer addressId) {
        AddressEntity address = addressService.get(addressId);
        StudentEntity student = studentRepository.findByAddress(address);
   }
```

JOIN – @Query 1

```
public interface StudentRepository ... {
     @Query("Select s from StudentEntity s where s.address.id =:id ")
    StudentEntity findByAddress(@Param("id") Integer addressId);
}

 public void getStudentById_2(Integer addressId) {
        StudentEntity student = studentRepository.findByAddress(addressId);
   }
```

```
Hibernate: select .... from student studentent0_ where studentent0_.address_id=?
```

JOIN – @Query 2

```
public interface StudentRepository ... {
    @Query("Select s from StudentEntity s INNER JOIN s.address a WHERE a.id=:id ")
   StudentEntity findByAddressJoin1(@Param("id") Integer addressId);
}
```

```
StudentEntity student = studentRepository.findByAddressJoin1(addressId);
```

```
Hibernate: select ...
 from student studentent0_
 inner join address addressent1_ on studentent0_.address_id=addressent1_.id
 where addressent1_.id=?
```

JOIN – @Query 3

```
public interface StudentRepository ...  {
    @Query("Select s from StudentEntity s, AddressEntity a WHERE a.id=:id ")
    List<StudentEntity> findByAddressNoJoinCondition(@Param("id") Integer addressId);
}
```

JOIN – @Query Native

```
public interface StudentRepository ... {
      @Query(value = "Select * from student AS s INNER JOIN address as a  " +
          " on s.address_id = a.id WHERE a.id =:id", nativeQuery = true)
   StudentEntity findByAddressNative(@Param("id") Integer addressId);
}
```

```
@OneToOne(fetch = FetchType.EAGER)
```

JOIN – OneToOne - reverse

- We Know Student and Address table and how they are joined .
- How to get Address if we know Student id ?
- How we can access to Address by knowing only Student Id ?

JOIN – OneToOne – mapped by

```java
@Entity
@Table(name = "student")
public class StudentEntity {
    @Id
    @GeneratedValue(…)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private Integer age;

    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "address_id")
    private AddressEntity address;
}
```

```java
@Entity
@Table(name = "address")
public class AddressEntity {
    @Id
    @GeneratedValue(…)
    private Integer id;
    @Column
    private String name;

    @OneToOne(mappedBy = "address", fetch = FetchType.LAZY)
    private StudentEntity student;
}
```

JOIN – OneToOne – mapped by query

```
public interface AddressRepository ... {
    AddressEntity findByStudent(StudentEntity studentEntity);

    @Query("Select a From AddressEntity a Where a.student.id =:id")
    AddressEntity findByStudentId(@Param("id") Integer studentId);
}
```

```
Hibernate: select ...
   from address addressent0_ cross join student studentent1_
   where addressent0_.id=studentent1_.address_id and
   studentent1_.id=?
```

JOIN – OneToOne – mapped by query Native

- In these case Mapped vashe not needed

```
public interface AddressRepository ... {

    @Query(value = "Select * From address a INNER JOIN student
                                    as s on s.address_id = a.id WHERE s.id =:id",
            nativeQuery = true)
    AddressEntity findByStudentIdNative(@Param("id") Integer studentId);
}
```

Continue …

- dasturlash.uz

ManyToOne JOIN Table

```
@Entity
@Table(name = "course")
public class CourseEntity {
    @Id
    @GeneratedValue(…)
    private Integer id;
    @Column
    private String name;
    @Column
    private Integer duration;
}
```

```
@Entity
@Table(name = "student_course")
public class StudentCurseEntity {
    @Id
    @GeneratedValue(…..)
    private Integer id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "student_id")
    private StudentEntity student;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "course_id")
    private CourseEntity course;

    @Column
    private LocalDateTime createdDate;
}
```

```
@Entity
@Table(name = "student")
public class StudentEntity {
    @Id
    @GeneratedValue(...)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private Integer age;
}
```

ManyToOne JOIN Example 1

```
@Query("Select s From StudentCurseEntity s Where s.student.id =:id")
  List<StudentCurseEntity> getByStudentId(@Param("id") Integer id);

  @Query("Select s From StudentCurseEntity s Where s.course.id =:id")
  List<StudentCurseEntity> getByCourseId(@Param("id") Integer id);
```

Hibernate: select ... from student_course studentcur0_ where studentcur0_.student_id=?

Hibernate: select….

ManyToOne JOIN Example 2

```
@Query("Select s From StudentCurseEntity s INNER JOIN s.student st " +
        "Where st.id =:id")
  List<StudentCurseEntity> getByStudentId(@Param("id") Integer id);
```

```
Hibernate: select ...
 from student_course studentcur0_ inner join student studentent1_
   on studentcur0_.student_id=studentent1_.id
        where studentent1_.id=?
```

Complex JOIN Example 1

- Get Student Course Joined Date by Student Id and Course Id

Complex JOIN Example 1 Solution

```
public interface AddressRepository ... {

  @Query("Select s From StudentCurseEntity s “
          “ INNER JOIN s.student st " +
          " INNER JOIN s.course c " +
          " Where st.id = :sId AND c.id =:cId")
    List<StudentCurseEntity> getByStudentIdAdnCourseId(@Param("sId") Integer sId,
                                                       @Param("cId") Integer cId);

}
```

```
from student_course studentcur0_
          inner join student studentent1_ on studentcur0_.student_id=studentent1_.id
          inner join course courseenti2_ on  studentcur0_.course_id=courseenti2_.id
 where studentent1_.id=? and courseenti2_.id=?
```

Mapping

- dasturlash.uz

Mapping get only some fields

- What if we want get only id and name from StudentEntity.
- We do not need other 15 fields.
- We need only id and name.
- What will be query.

```
@Entity
@Table(name = "student")
public class StudentEntity {
    @Id
    @GeneratedValue(…)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private Integer age;

    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "address_id")
    private AddressEntity address;
}
```

Mapping get only some fields example

- We can not sol it with method name query.
- We need user @Query (JPQL or SQL native) query.

```
@Query("Select new StudentEntity(p.id, p.name) FROM StudentEntity p WHERE p.id =:id")
StudentEntity  findNameById(@Param("id") Long id);
```

```
@Query("Select new com.company.entity.StudentEntity(p.id, p.name) FROM StudentEntity p
                        WHERE p.id =:id")
StudentEntity findNameById(@Param("id") Long id);
```

- dasturlash.uz

Complex Mapping

- Get only Student id, name and addressName by student id. Other fields not needed.

```
@Entity
@Table(name = "student")
public class StudentEntity {
    @Id
    @GeneratedValue(…..)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private Integer age;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "address_id")
    private AddressEntity address;
}
```

```
@Getter
@Setter
@Entity
@Table(name = "address")
public class AddressEntity {
    @Id
    @GeneratedValue(…….)
    private Integer id;
    @Column
    private String name;
}
```

Complex Mapping solution 1

- Create DTO class for mapping with exactly same parametrs as query result.
- Create all parametr constructor.

```
@Query("Select new com.company.mapper.StudentNameAndContactDTO(s.id,s.name,a.name) " +
        " from StudentEntity s INNER JOIN s.address a WHERE s.id=:id ")
StudentNameAndContactDTO findStudentNameAndAddressName(@Param("id") Integer studentId);
```

```
public class StudentNameAndContactDTO {
    private Integer id;
    private String name;
    private String addressName;

    public StudentNameAndContactDTO(Integer id, String name, String addressName) {
        this.id = id;
        this.name = name;
        this.addressName = addressName;
    }
}
```

Complex Mapping solution using interface

```
public interface IStudentNameAndContactDTO {
    Long getId();
    String getName();
    String getAddressName();
}
```

```
@Query("Select s.id as id, s.name as name , a.name as addressName " +
        " from StudentEntity s INNER JOIN s.address a WHERE s.id=:id ")
 IStudentNameAndContactDTO findStudentNameAndAddressNameInter(@Param("id") Integer studentId);
```

```
IStudentNameAndContactDTO student = studentRepository.findStudentNameAndAddressNameInter(id);
System.out.println(student.getId()+"" + student.getName() +" "+ student.getAddressName());
```

- As required

- dasturlash.uz

- Native queryda Enumblarni String sifatida berish kerak.

Make new slides form this

- @SqlResultSetMapping(name="groupDetailsMapping", classes={ @ConstructorResult(targetClass=GroupDetails.class, columns={ @ColumnResult(name="GROUP_ID"), @ColumnResult(name="USER_ID") }) }) @NamedNativeQuery(name="getGroupDetails", query="SELECT g.*, gm.* FROM group g LEFT JOIN group_members gm ON g.group_id = gm.group_id and gm.user_id = :userId WHERE g.group_id = :groupId", resultSetMapping="groupDetailsMapping")

Mapping Links

- https://www.baeldung.com/jpa-queries-custom-result-with-aggregation-functions
- https://thorben-janssen.com/spring-data-jpa-dto-native-queries/