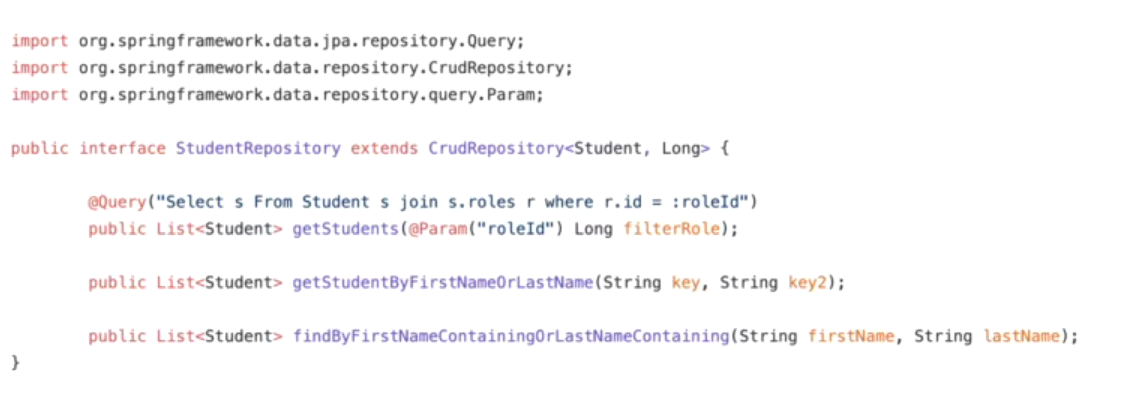


Toate interfetele sunt tranzactionale

* Totusi, daca de ex metoda save() de la JpaRepository e chemata intr-o tranzactie, ea nu va mai folosi tranzactia proprie, ci va folosi tranzactia metodei de unde e apelata
* 

@Param e cel de dupa : din Query

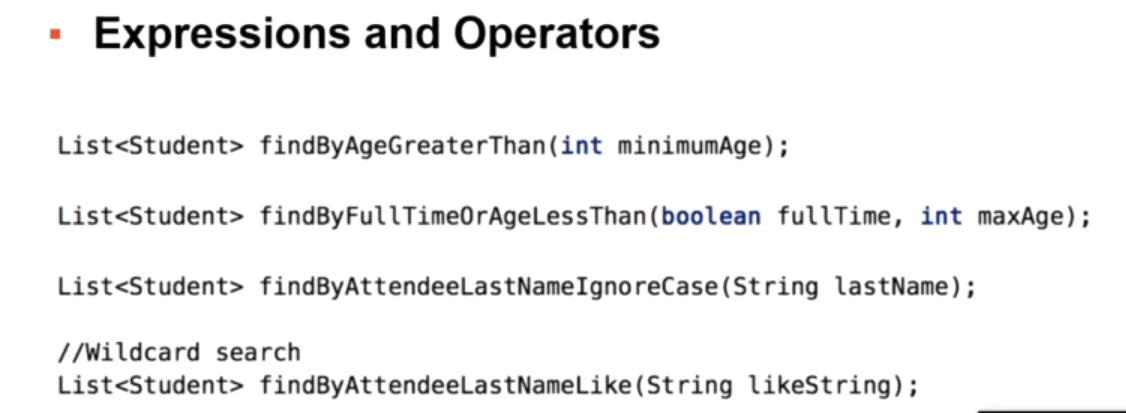
* orice metoda ce face un update necesita anotatia @Modifing, de altfel vom avea exceptie la runtime
* JpaRepository;

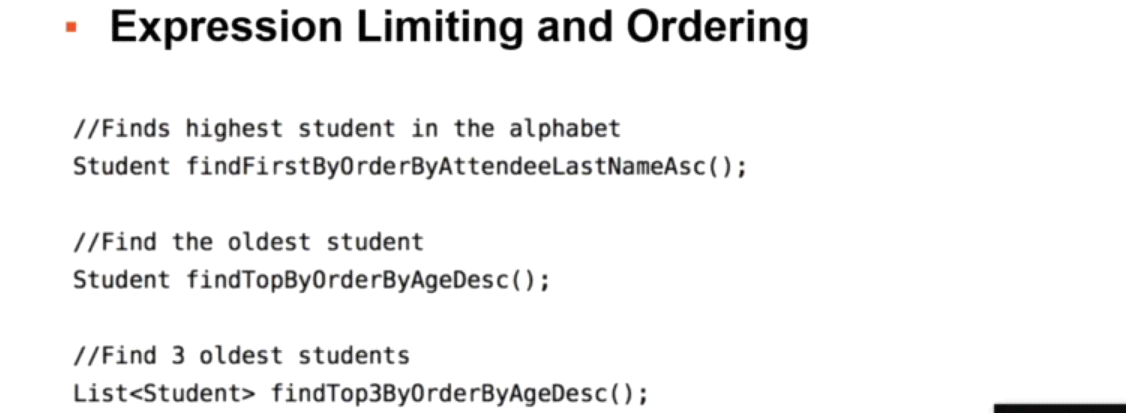


JPA salveaza datele in baza de date cat mai tarziu. save() nu inseamna ca datele vor fi scrise imediat in baza de date, ci mai degraba la final de tranzactie.

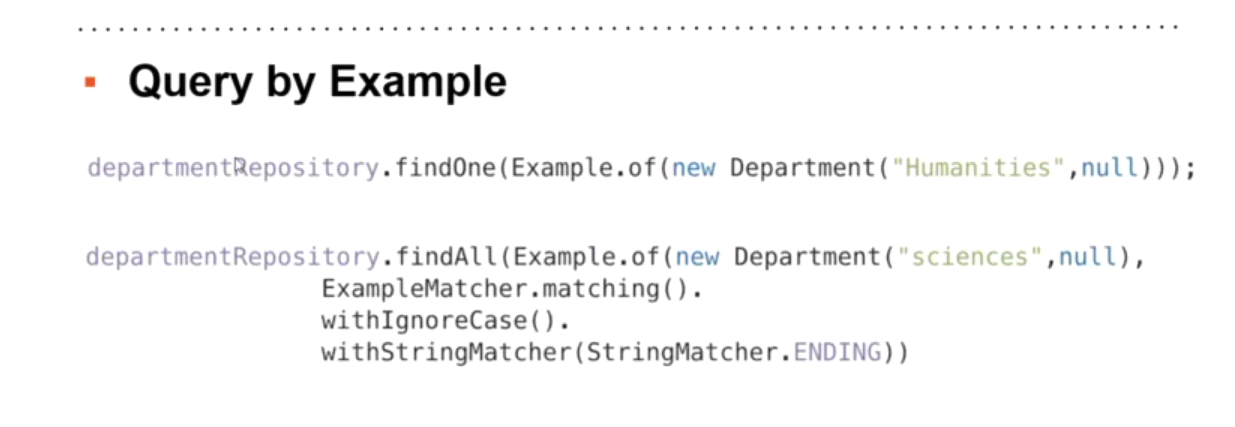
**flush()** – face ca queries sa fie executate in baza de date, dar sa nu fie commit inca. E util cand gen vrem sa primit un id de la un user si avem envoie de el tot in tranzactie pana a da commit.

**deleteInBatch()** – deleteAll face un forEach() pentru fiecare item. deleteInBatch creaza un singur query pentru a sterge toate datele

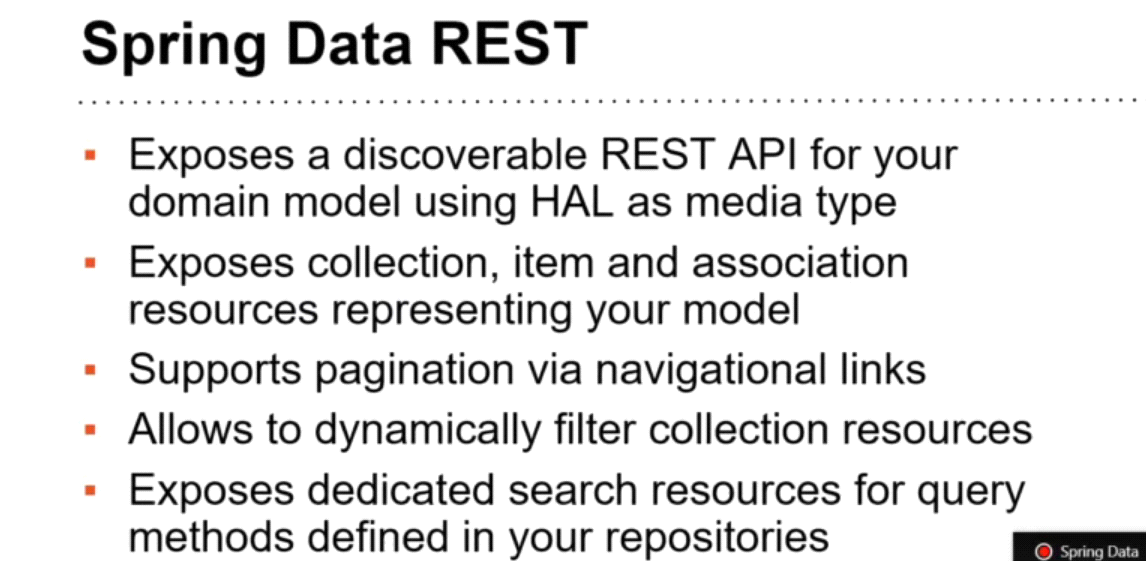
* Spring mereu creaza un object proxy pentru repository creat cu noi de JPA si de asta putem da @Autowired
* 

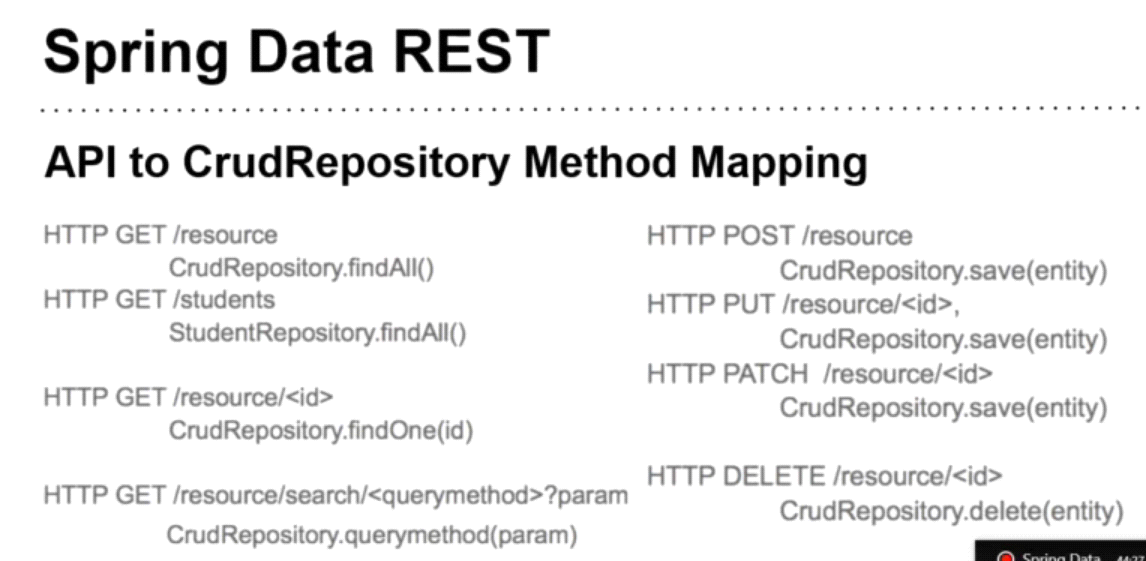


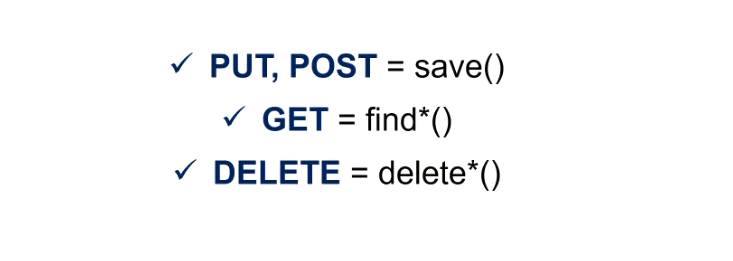




**Spring Data REST**







* Spring Data REST urmeaza HATEOAS si HAL
* **spring.data.rest.basepath=/api** – vom folosi /api cu orice endpoints din rest

**Hide repsitory**

* Faptul ca Spring REST Data creaza endpoints pentru fiecare entity nu e deloc bine
* Pentru a rezolva problema, putem pune o proprietate in application.properties ce nu va crea endpointuri pentru niciun entity automat, ci le vom seta noi:

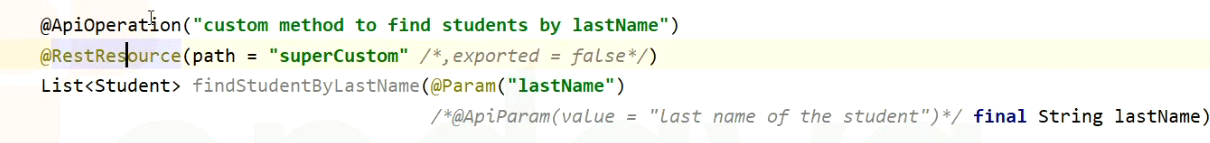
**spring.data.rest.detection-strategy=annotated**

Asa, se vor crea doar pentru entity anotate cu **@RepositoryRestResource**

* **@NoRepositoryBean** – asta spune ca acea interfata nu trebuie tratata ca un Repository de Spring Data, chiar daca exitende JpaRepository sau CrudRepository sau alta

**@RestResource**

* Anotatia se pune deasupra la metoda pentru a modifica endpoint:



sau pentru a dezactiva anume endpoints. Deoarece JpaRepository are metoda deleteById, logic ca avem si endpoint pentru DELETE /students/{id}

Pentru a o dezactiva, pur si simplu o suprascriem si folosim @RestResource(exported=false)

@RepositoryRestResource(path = "courses")  
public interface CourseRepository extends JpaRepository<Course,Integer> {  
 @Override  
 @RestResource(exported = false)  
 void deleteById(Integer integer);  
}

- @ApiOperation ofera descriere la ce face metoda

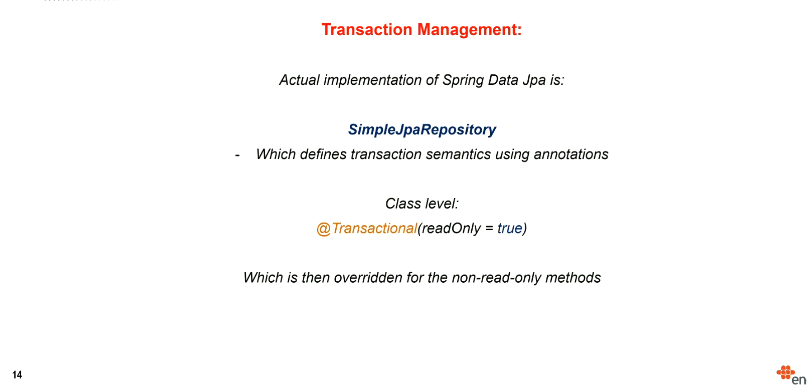
**Modify endpoint**

* Punem deasupra la clasa @Repository asta:

@RepositoryRestResource(path=”entity”)

**@Modifying**

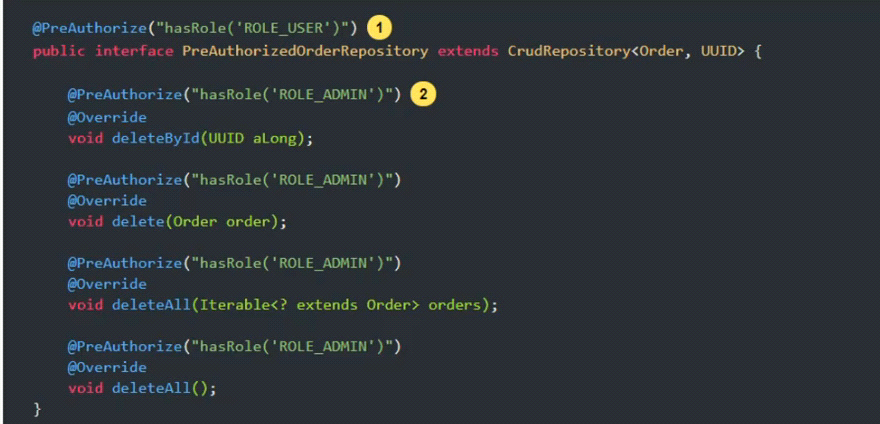
* Cand avem metode ce modifica datele din baza de date, e necesar de a pune @Modifying
* Totodata, va trebui si @Transactional, deoarece proxy creat nu va contine transactia ce permite si modificarea datelor;
* @Query("UPDATE Teacher t SET t.firstName=:fn WHERE t.id=:id")  
  @Modifying  
  @Transactional  
  void updateFirstNameById(@Param("fn")String firstName,@Param("id")int id);



Iata de ce, ne trebuie @Modifying, caci e pus readOnly=true la @Transactional

In mod normal, orice metoda din JpaRepository sau altele e doar pentru a citi date, si asta din cauza ca asa e pus la SimpleJpaRepository class

**Spring Security @PreAuthorize(‘hasRole(“rol”)’)**



merge si la nivel de clasa si apoi de metoda

**Projection**

* Uneori s-ar putea sa nu avem nevoie de toate fieldurile unui entity returnat in baza de date
* De ex, Course are si title, si hours si id si multe altele poate avea, dar poate noua ne trebuie doar titlurile cursurilor returnate
* O soluti usoara este de a crea o interfata, si de a defini metodele getter pentru fieldurile ce vrem sa fie returnate. De ex, fie repository:
* @Getter  
  @Setter  
  @ToString  
  @NoArgsConstructor  
  @Entity  
  @Table(name = "course")  
  public class Course {  
   @Id  
   @Column(name = "id")  
   @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
   private int id;  
   @Column(name = "title")  
   private String title;  
   @Column(name = "hours")  
   private String hours;  
   @ManyToOne  
   @JoinColumn(name = "teacher\_id")  
   private Teacher teacher;  
    
   public Course(String title, String hours, Teacher teacher) {  
   this.title = title;  
   this.hours = hours;  
   this.teacher = teacher;  
   }  
  }

Aici avem si metoda getTitle()

Acum, daca am vrea sa obtinem o lista doar cu numele cursurilor, facem o interfata ce contine doar o metoda getTitle()

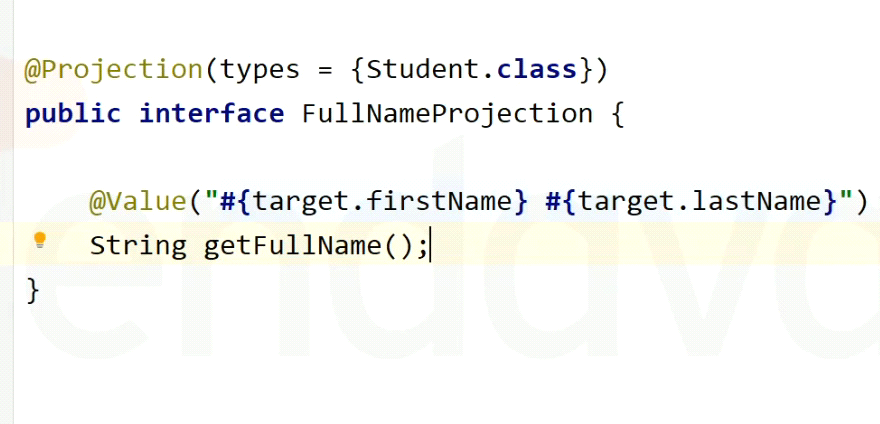
public interface CourseTitle {  
 String getTitle();  
}

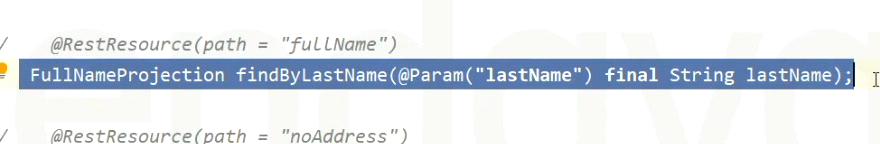
Si, cand vom crea metoda in Repository interface, nu vom returna o List<Course>, ci o List<CourseTitle>.

@RepositoryRestResource(path = "courses")  
public interface CourseRepository extends JpaRepository<Course,Integer> {  
 List<CourseTitle> getCourseByTeacherId(int id);  
}

Spring va crea un proxy pentru CourseTitle cand va vedea ca folosim clasa CourseTitle ca tip de return si va returna deja nu o Lista de Course, ci o lista de CourseTitle. Defapt, el va vedea ca tipul listei nu e Course, si va vaea grija sa creeze o clasa cu field necesar ce va contine doar numele. Nici nu va crea obiect de tip Course, ci CourseTitle

**@Projection**

* 



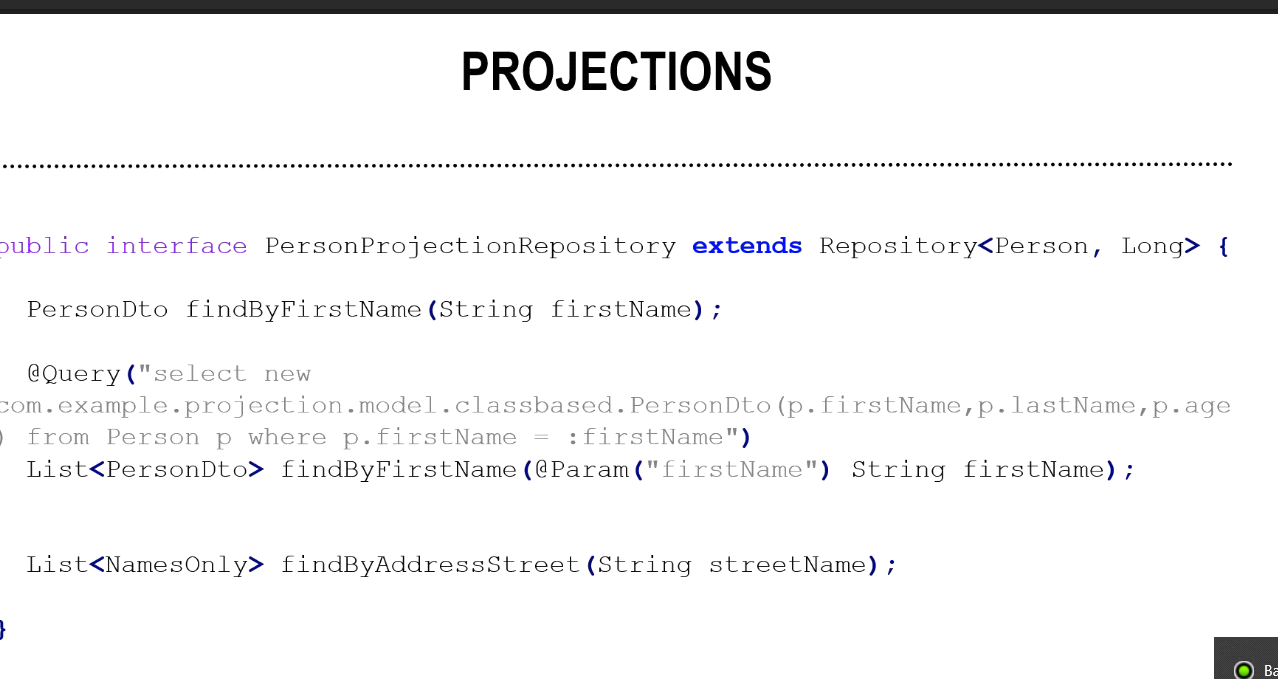
Returneaza doar coloanele firstName si lastName din tabel

Deci, putem folosi si anotatia @Value(“target.numeField”) pentru a spune exact cum se numesc coloanele ce le vrem si le putem combina. target. e cel ce pastreaza coloanele

@Projection(types = {Course.class})  
public interface CourseTitle {  
 @Value("#{target.title}")  
 String getTitleMethod();  
}

**Class Projection**

* Trebuie sa aiba doar un construtor cu fieldurile



**@NamedQuery**

* Ne permite sa definim query chiar in clasa @Entity, ca sa nu trebuiasca sa le definim in Repository class
* @NamedQuery(name = “Clasa.NumeLaQuery”,”Query”)

Atentie ca numele e Clasa.NumeQuery!

@Entity

@NamedQuery(name = "Teacher.getAllRecordsOrdered",query = "FROM Teacher ORDER BY first\_name")  
public class Teacher {

* Apoi in repository pur si simplu folosim numele acestui query fara a mai specifica vreun @Query

public interface TeacherDAO extends CrudRepository<Teacher,Integer> {  
 List<Teacher> getAllRecordsOrdered();  
}

* Putem folosi si @NamedQueries pentru a specifica mai multe:

@NamedQueries({  
 @NamedQuery(name = "Teacher.getAllRecordsOrdered",query = "FROM Teacher ORDER BY first\_name"),  
 @NamedQuery(name = "Teacher.someQury",query = "FROM Teacher WHERE id = 21")}  
)

**Streamable<>**

* Spring Data ofera si posibilitatea de a returna un Stream direct:
* public interface TeacherDAO extends CrudRepository<Teacher,Integer> {  
   Streamable<Teacher> getAllRecordsOrdered();  
  }

**CriteriaBuilder**

**Events Spring Data**

**Views**

* In view nu pot fi inserate date, de aceea crearea la un repository nu e o idee prea buna, asa cum se ofera si metode de a insera si sterge datele
* Spring DATA ofera o solutie pentru asta. Cream o simpla interfata ce extidente o clasa de la Spring Data, ca Repository<>, ce nu are metode,dar o anotam cu @NoBeanRepsitory si adaugam metodele noastre, apoi cream o alta interfata ce extinde aceasta interfata si gata, Spring Data va vedea ca ea extinde si o interfata de a lui, si va crea repository, si asa cum Repository<> nu are metode, se vor crea doar metodele noastre si gata,