Spring Data

Spring Data Commons

1. Java business entities ⇄ Persistent target datastore records
2. Lookup records
3. Update records
4. Delete records

Repository Pattern

- Repository pattern
- Create, read, update, and delete (CRUDRepository)
- JpaRepository
- MongoRepository

Object-Relational Mapping (ORM)

- Physical model to the logical model
- Physical model = Relational database
- Logical model = Java domain objects

ORM with Standard Java

1. Open a transaction

2. Make a SQL query

3. Iterate through each record

4. Iterate through each field in a record

ORM with Standard Java

5. Extract field, respecting data type

6. Map to the Java object/attribute

7. Close the transaction

7a. For Insert/Update query – commit/rollback transaction

Java Persistence API: Backstory

Mid-2000s: Hibernate, TopLink, and IBATIS

Code divergence

Sun Microsystems + Industry leaders

Java Community process => JSR 317 => JPA 2.0

December 2009

JPA Is Just a Specification

- **Implementation frameworks: Hibernate, TopLink, and Java EE application servers**
- **Metadata mapping (XML or Java annotations)**

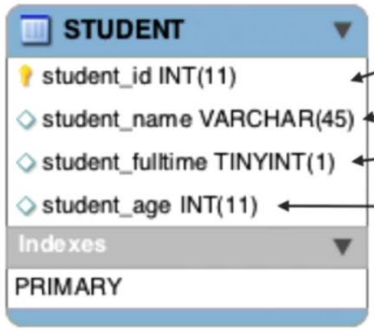
  Java entities <==> Tables

  Java attributes <==> Column/fields

- **EntityManager**

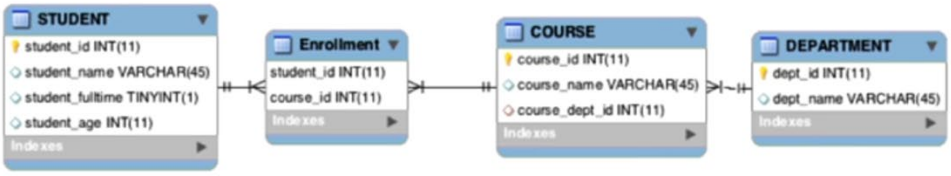
  Create, read, update, and delete entities

Map a Database Table to a Java Class

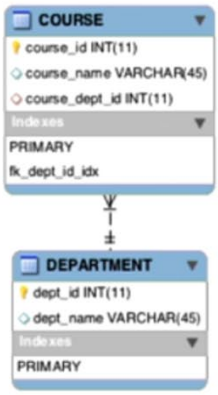



```
@Entity
@Table(name="STUDENT")
public class Student {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   @Column(name="student_id")
   private Integer studentId;

   @Column(name="student_name")
   private String name;

   @Column(name="student_fulltime")
   private boolean fullTime;

   @Column(name="student_age")
   private Integer age;

   public Student(String name, boolean fullTime,
           Integer age) {
     this.name = name;
     this.fullTime = fullTime;
     this.age = age;
   }
  //getters and business logic here
```

Map Multiple Tables to Java Classes

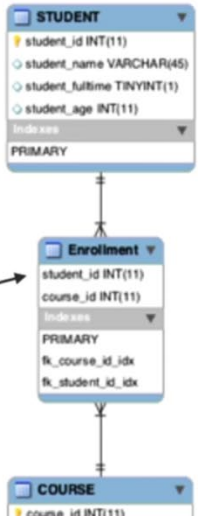



```
@Entity
@Table(name="Course")
public class Course {
  @Id
  @GeneratedValue
  @Column(name="course_id")
  private Integer id;

  @Column(name="course_name")
  private String name;

  @ManyToOne
  @JoinColumn(name="course_dept_id")
  private Department department;

  public Course(String name,
          Department department) {
    this.name = name;
    this.department = department;
  }
 @Override
 public String toString() {
   return "Course{" +
    "id=" + id + ", name='" + name + '\'' +",
    department=" + department.getName() + '}';

 }
...}
```




```
@Entity
@Table(name="Department")
public class Department {
  @Id
  @GeneratedValue
  @Column(name="dept_id")
  private Integer id;

  @Column(name="dept_name")
  private String name;

  @OneToMany(mappedBy="department",
                fetch=FetchType.EAGER,
                cascade = CascadeType.ALL)
  private List<Course> courses = new
ArrayList<>();

  public Department(String name) {
    this.name = name;
  }

 @Override
 public String toString() {
   return "Department{" + "id=" + id +",
name='" +
     name + '\'' +  ", courses=" + courses +'}';
}
...}
```

```
@Entity
@Table(name="STUDENT")
public class Student {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="student_id", nullable = false)
private Integer studentId;

@Column(name="student_name", nullable = false)
private String name;

@Column(name="student_fulltime", nullable = false)
private boolean fullTime;

@Column(name="student_age")
private Integer age;

@OneToMany(fetch = FetchType.EAGER,cascade =
CascadeType.ALL)
@JoinTable(name="Enrollment",
    joinColumns = {@JoinColumn(name = "student_id")},
    inverseJoinColumns = {@JoinColumn(name =
"course_id")} )
private List<Course> courses = new ArrayList<>();
```



Java Persistence Query Language (JPQL)

- Interact with entities and their persistent state
- Portable to any database management system
- Syntax similar to SQL
- JPQL – entities and attributes
- SQL – tables and columns

Now with JPQL

```
@PersistenceContext
private EntityManager entityManager;
public void printJane() {
  Student jane =  entityManager.createQuery("Select s from Student s where s.name='jane'", Student.class).getSingleResult();
  System.out.println(jane);
}
```

```
{
studentId=1, name='jane', fullTime=true, age=40, courses=
     [Course{id=2, name='chemistry', department='Science'},
      Course{id=3, name='physics', department='Science'},
      Course{id=4, name='compsci', department='Science'}
     ]
}
```

Lazy Initialization Exception

```
@OneToMany(cascade = CascadeType.ALL)
@JoinTable(name="Enrollment",
       joinColumns = {@JoinColumn(name = "student_id")},
       inverseJoinColumns = {@JoinColumn(name = "course_id")} )
private List<Course> courses = new ArrayList<>();
```

```
Caused by: org.hibernate.LazyInitializationException: failed to lazily initialize a
collection of role: com.example.university.Student.courses, could not initialize proxy
- no Session
```

JPA without Spring Data

Create

```
@PersistenceContext
EntityManager entityManager;

Student create(String name, boolean isFullTime, int age) {
  entityManager.getTransaction().begin();
  Student newStudent = new Student(name, isFullTime, age);
  entityManager.persist(newStudent);
  entityManager.getTransaction().commit();
  entityManager.close();
  return  newStudent;
}
```

Update

```
@PersistenceContext
EntityManager entityManager;

void updateAge(int studentId, int age) {
  entityManager.getTransaction().begin();
  Student student = entityManager.find(Student.class, studentId);
  student.setAge(age);
  entityManager.persist(student);
  entityManager.getTransaction().commit();
  entityManager.close();
}
```

Delete

```
@PersistenceContext
EntityManager entityManager;

void delete(int studentId) {
  entityManager.getTransaction().begin();
  Student student = entityManager.find(Student.class, studentId);
  entityManager.remove(student);
  entityManager.getTransaction().commit();
  entityManager.close();
}
```

Read/Lookup

```
@PersistenceContext
EntityManager entityManager;

List<Student> read(String nameLike) {
  Query query = entityManager.createQuery(
  "select s from Student s where s.name LIKE :someName", Student.class);
  query.setParameter("someName", "%" + nameLike+ "%");
  List<Student> result = query.getResultList();
  entityManager.close();
  return result;
}
```

Spring Data Repository Interfaces

Spring Data Repository Interfaces

```
public interface Repository<T, ID>

T        Domain type the repository manages
ID       Type of the entity id
```

Spring Data Repository Interfaces

```
public interface CrudRepository<T, ID> extends
Repository<T, ID>

package com.springframework.data.repository
```

Create/Update Methods

```
T save(T entity);

Iterable<T> saveAll(Iterable<T> entity);
```

Delete Methods

Spring Data V2

```
void deleteById(ID id)
void deleteAll(Iterable<? extends T>)
void delete(T var1)
void deleteAll()
```

Spring Data V1.X

```
void delete(ID id)
```

Read Methods

Spring Data V2

```
Optional<T> findById(ID id)
Iterable<T> findAllById(Iterable<ID> ids)
Iterable<T> findAll()
long count()
boolean existsById(ID id)
```

Spring Data V1.X

```
T findOne(ID id)
Iterable<T> findAll(Iterable<ID> ids)
boolean exists(ID id)
```

Student CrudRepository

```
public interface StudentRepository extends CrudRepository<Student,
Integer>
```

JPA Repository

Spring Data Store-Specific Interfaces

JpaRepository

```
public interface DepartmentRepository extends JpaRepository<Department, String> {
}
```

All features of CrudRepository plus:

- void flush();
- Department saveAndFlush(Department department);
- void deleteInBatch(Iterable<Department> iterable);
- void deleteAllInBatch();

Benefit

- No need to access EntityManagerFactory

```
@PersistenceUnit
EntityManagerFactory emf;
```

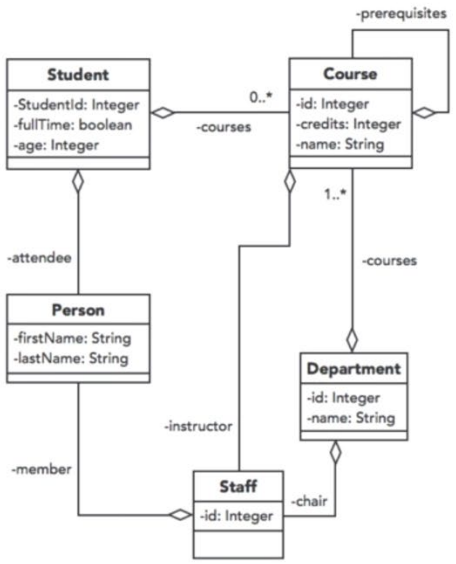
- Differentiate from other data repositories

  MongoRepository: MongoDB

  SolrCrudRepository: Apache Solr

  GemfireRepository: Pivotal GemFire

Querying with Spring Data



Repository Interfaces

- StudentRepository
- CourseRepository
- DepartmentRepository
- StaffRepository

Person is embeddable, not entity.
No PersonRepository.

Simple Query Method Property Expression Rules

1. Return type
2. findBy
3. Entity attribute name (use camel case)
4. Optionally, chain subattribute names (i.e., findByAttendeeLastName)
5. Parameter with datatype of the entity attribute

Unlike regular SQL, syntax errors are found at startup, not runtime.

```
List<Student> findByAtendeLastName(String name)

PropertyReferenceException: No property atendeLastName found
for type Student!
```

Query Method: Conditional Expressions

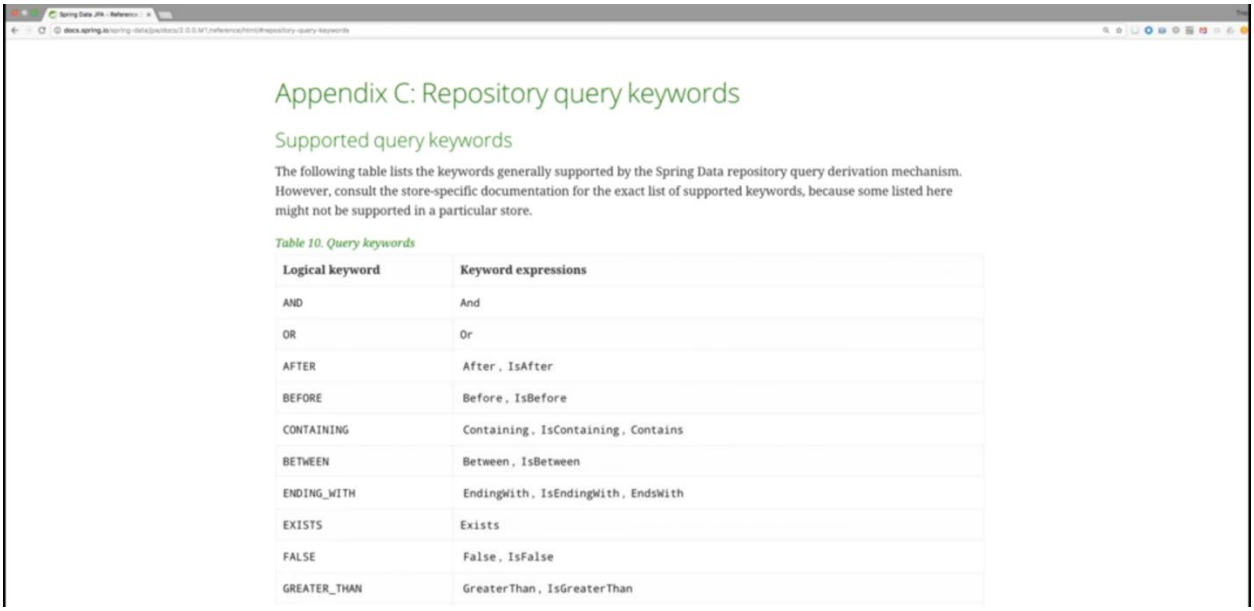
```
List<Student> findByFullTimeOrAgeLessThan(boolean fullTime, int maxAge);

List<Student> findByAttendeeFirstNameAndAttendeeLastName
                                    (String firstName, String lastName);

//Same results
List<Student> findByAttendee(Person person);
```

Query Method: Expressions with Operators

```
List<Student> findByAgeGreaterThan(int minimumAge);

List<Student> findByFullTimeOrAgeLessThan(boolean fullTime, int maxAge);

List<Student> findByAttendeeLastNameIgnoreCase(String lastName);

//Wildcard search
List<Student> findByAttendeeLastNameLike(String likeString);
```

Query Method: Expression Limiting and Ordering

```
//Finds highest student in the alphabet
Student findFirstByOrderByAttendeeLastNameAsc();

//Find the oldest student
Student findTopByOrderByAgeDesc();

//Find 3 oldest students
List<Student> findTop3ByOrderByAgeDesc();
```



Appendix C: Repository query keywords

Supported query keywords

The following table lists the keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some listed here might not be supported in a particular store.

Table 10. Query keywords

Logical keyword	Keyword expressions
AND	And
OR	Or
AFTER	After, IsAfter
BEFORE	Before, IsBefore
CONTAINING	Containing, IsContaining, Contains
BETWEEN	Between, IsBetween
ENDING_WITH	EndingWith, IsEndingWith, EndsWith
EXISTS	Exists
FALSE	False, IsFalse
GREATER_THAN	GreaterThan, IsGreaterThan

Using @Query Annotation on method

@Query-Annotated Query Method

```
@Query("JPQL query string")
ReturnValue anymethodname(zero or more parameters);

@Query(value="SQL query string", nativeQuery=true)
ReturnValue anymethodname(zero or more parameters);
```

When we should use @Query annotated method

Cleaner Method Signature

```
public interface CourseRepository extends CrudRepository<Course,String> {

   List<Course> findByDepartmentChairMemberLastName(String chairLastName);

   @Query("Select c from Course c where c.department.chair.member.lastName=:chair")
   List<Course> findByChairLastName(@Param("chair")String chairLastName);
}
```

Numbered query parameters

```
 @Query("Select c from Course c where c.department.chair.member.lastName = ?1")
   List<Course> findByChairLastName(String chairLastName);
```

Complex JPQL Queries

```
public interface CourseRepository extends CrudRepository<Course,String> {

  @Query("Select c from Course c join c.prerequisites p where p.id = ?1")
  List<Course> findByPrerequisite(int id);

  @Query("Select new com.example.university.CourseView(c.name,
          c.instructor.member.lastName, c.department.name) from Course c where c.id=?1")
  CourseView getCourseView(int courseId) ;
}
```

Going Native

```
public interface StudentRepository extends CrudRepository<Course,String> {

 @Query(value="SELECT * FROM student s ORDER BY s.student_age LIMIT 3", nativeQuery=true)
 List<Student> findThreeYoungestStudents();
}
```

Paging and Sorting

Create Paging and Sorting Query Method

```
public interface CourseRepository extends CrudRepository<Course,String> {

List<Course> findByCredits(int credits);
Page<Course> findByCredits(int credits, Pageable pageable);

};


Look up the first four courses that are three credits, sort them by
credits, and name
courseRepository.findByCredits(3,PageRequest.of(0,4,Sort.Direction.ASC,
"credits","name"));
0 = The page number (zero is the first page)
4 = size of the page
```

PagingAndSortingRepository Extends CrudRepository

```
public interface StaffRepository extends PagingAndSortingRepository<Staff,Integer> {
}

Available methods
Iterable<Staff> findAll(Sort sort);

Page<Staff> findAll(Pageable pageable);


Iterable<Staff> allStaffSortedByFirstname =
  staffRepository.findAll(new Sort(Sort.Direction.ASC,"member.firstName"));

Page<Staff> first5StaffMembersSortedByLastName = staffRepository.findAll(
  PageRequest.of(0,5,new Sort(Sort.Direction.ASC,"member.lastName")));
```

QueryBy Example

Query by Example

- User-friendly alternative to SQL
- Lookup objects similar to another object
- Independent of underlying datastore

Query by Example

- Frequently refactored code
- Code requiring nested property constraints or complex string matching

```
JpaRepository<Department, Integer>
                extends QueryByExampleExecutor<Department>

public interface DepartmentRepository extends JpaRepository<Department, String> {
}
```

Available methods

```
List<Department> findAll(Example<Department> example);

List<Department> findAll(Example<Department> example, Sort sort);

Optional<Department> findOne(Example<Department> example);

Page<Department> findAll(Example<Department> example, Pageable pageable);

long count(Example<Department> example);

boolean exists(Example<Department> example);
```

Example<T> example = Example.of(T probe);

```
Given the following Constructors:
 Department(String name, Staff chair)
 Staff(Person member)
 Person(String firstName, String lastName)

Find the department with the name “Humanities”:

   departmentRepository.findOne(Example.of(new Department("Humanities",null)));


Find all departments whose chair has the first name of ”John”:

   departmentRepository.findAll(Example.of(new Department(null,new Staff(new
Person("John",null)))));
```

Example<T> example = Example.of(T probe, ExampleMatcher matcher);

```
Find all departments with the name ending in sciences; ignore case:

departmentRepository.findAll(Example.of(new Department("sciences",null),
                ExampleMatcher.matching().
                withIgnoreCase().
                withStringMatcher(StringMatcher.ENDING))
```

Optional<> query Response

```
@Query("Select new com.example.university.view.CourseView" +
        "(c.name, c.instructor.member.lastName, c.department.name) from Course c where c.name=?1")
Optional<CourseView> getCourseViewByName(String name);
```

More Repository Types

1. **MongoRepository**

```
public interface DepartmentRepository extends MongoRepository<Department, String> {

    Optional<Department> findByName(String name);

    @Query("{ 'name' : { $regex: ?0 } }")
    List<Department> findNameByPattern(String pattern);

    //This method fails because cannot perform Joins across DBRef's
    List<Department> findByChairMemberLastName(String lastName);

}
```

2. **Spring Data JDBC**

Spring Data JDBC Repository
Relational DBMS without JPA

Pros and Cons of JPA

Java Persistence API (Hibernate)

Pros

- Lazy loading, caching, dirty tracking

Cons

- Expensive SQL statements and unexpected exceptions
- External database updates not in cache
- Point of operator persistent not obvious

Pros and Cons of JDBC

Spring Data JDBC Repositories

Pros

- Simpler model, SQL issued when needed, fully loaded object

Cons

- Many-to-one and many-to-many relationships not supported

Maven Dependency for jdbc

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Chair Table in JDBC

```
@Table("CHAIR")
public class Chair {
    @Id
    private int department;

    private String name;

    public Chair(String name) { this.name = name; }

    @Override
    public String toString() {
        return "Chair{" +
                "department=" + department +
                ", name='" + name + '\'' +
                '}';
    }
}
```

Unlike as JPA which creates table automatically on runtime . For JDBC we will be using schema.sql scripts to generate the tables .

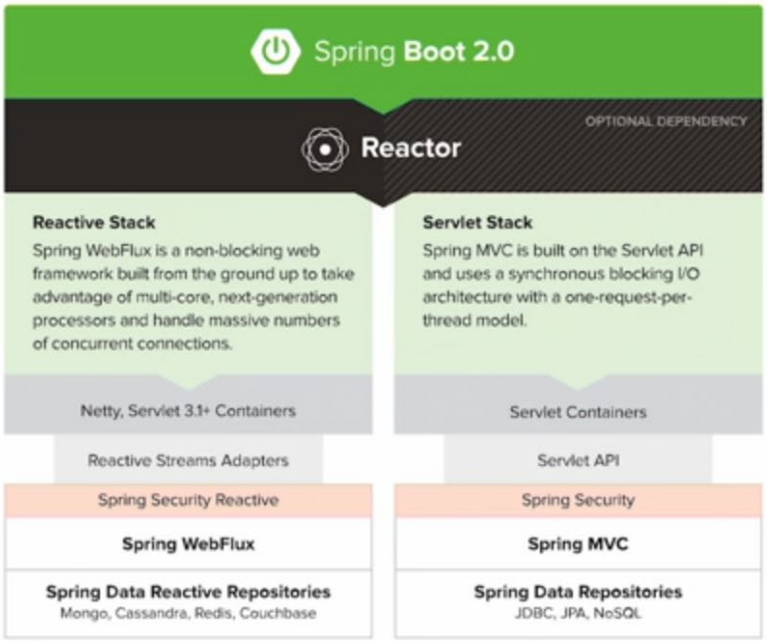
Schema Sql

```
CREATE TABLE DEPARTMENT (
  ID INTEGER AUTO_INCREMENT PRIMARY KEY,
  NAME varchar(100) NOT NULL
);

CREATE TABLE CHAIR (
  DEPARTMENT INTEGER REFERENCES DEPARTMENT(ID),
  NAME varchar(100) DEFAULT NULL
);
```

Department Repository

```
public interface DepartmentRepository extends CrudRepository<Department, String> {

    @Query("SELECT DEPARTMENT.id AS id, DEPARTMENT.name AS name, chair.department AS chair_department, chair.name AS chair_
            "FROM DEPARTMENT LEFT OUTER JOIN CHAIR AS chair ON chair.DEPARTMENT = DEPARTMENT.id " +
            "WHERE DEPARTMENT.name =:name")
    Optional<Department> findByName(@Param("name")String name);
}
```

Reactive Repository



THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND

Developers are constantly challenged with choosing the most effective runtime, programming model, and architecture for their application's requirements and team's skill set. For example, some use cases are best handled by a technology stack based on synchronous blocking I/O architecture, whereas others would be better served by an asynchronous, nonblocking stack built on the reactive design principles described in the Reactive Streams Specification.

Reactive Spring represents a platform-wide initiative to deliver reactive support at every level of the development stack: web, security, data, messaging, etc. Spring Framework 5 delivers on this vision by providing a new reactive web stack called Spring WebFlux, which is offered side by side with the traditional Spring MVC web stack. The choice is yours!

Maven Dependency for reactive mongo

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Department repository extending ReactiveCrudRepository

```
public interface DepartmentRepository extends ReactiveCrudRepository<Department, String> {

}
```

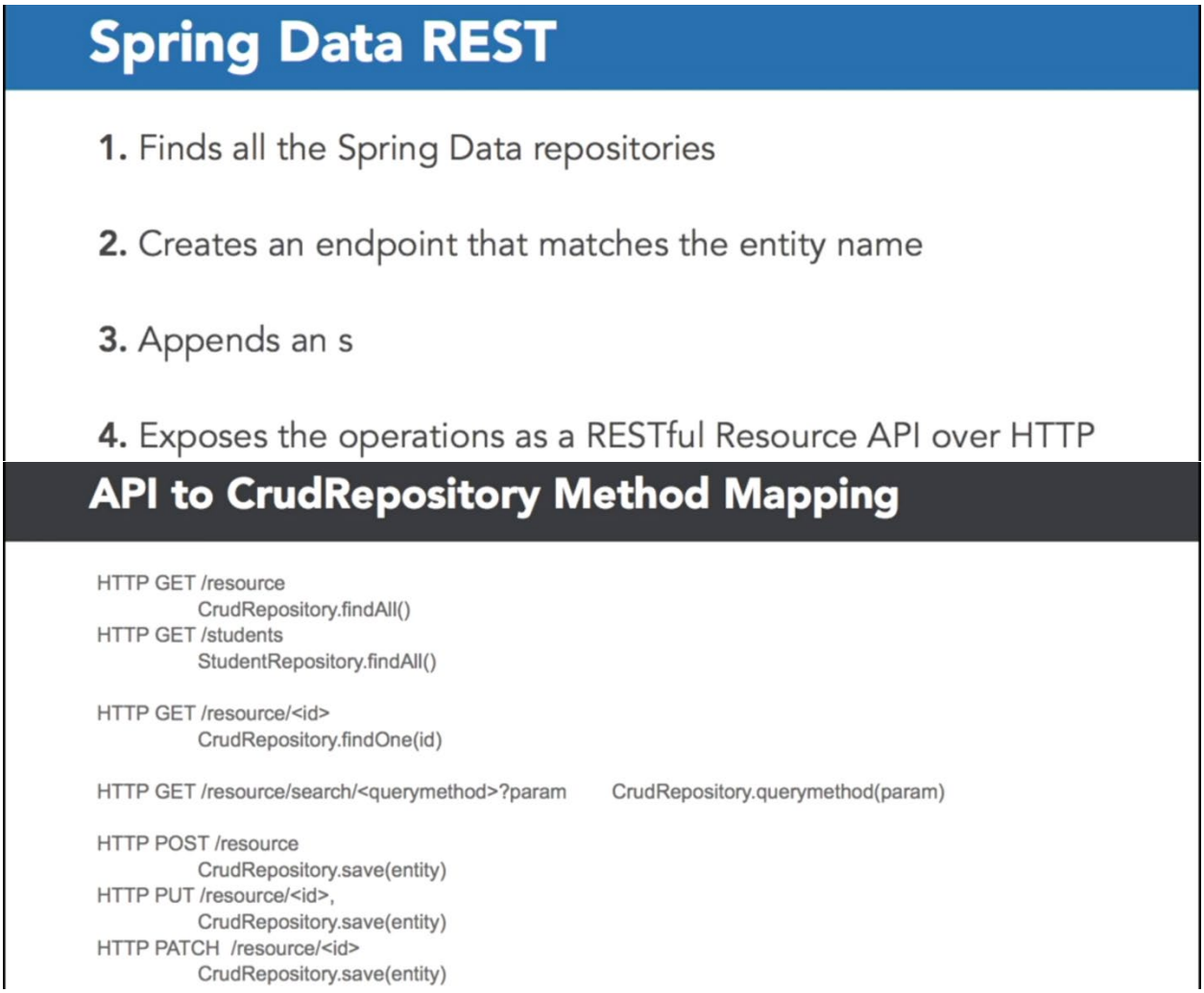
Staff repository

```
public interface StaffRepository extends ReactiveCrudRepository<Staff, Integer> {
    Flux<Staff> findByMemberLastName(String lastName);
}
```

Other methods of ReactiveCrudRepository

```
@NoRepositoryBean
public interface ReactiveCrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> Mono<S> save(S var1);

    <S extends T> Flux<S> saveAll(Iterable<S> var1);

    <S extends T> Flux<S> saveAll(Publisher<S> var1);

    Mono<T> findById(ID var1);

    Mono<T> findById(Publisher<ID> var1);

    Mono<Boolean> existsById(ID var1);

    Mono<Boolean> existsById(Publisher<ID> var1);

    Flux<T> findAll();

    Flux<T> findAllById(Iterable<ID> var1);

    Flux<T> findAllById(Publisher<ID> var1);
```

Spring Data REST



Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Querydsl Extension

Query Methods Are Static

```
public interface StudentRepository extends CrudRepository<Student, Integer> {
   List<Student> findByAttendeeLastName(String lastName);
   List<Student> findByFullTime(Boolean isFullTime);
   List<Student> findByAgeGreaterThan(int age);
   List<Student> findByAttendeeLastNameAndFullTime(String lastName,
                                    boolean isFullTime);
   List<Student> findByAttendeeLastNameAndAgeGreaterThan(String lastName,
                                                         int age);
   List<Student> findByAttendeeLastNameAndAgeGreaterThanAndFullTime(
                               String lastName, int age, boolean isFullTime);
   List<Student> findByFullTimeAndAgeGreaterThan(Boolean isFullTime, int age);
}
```

Querydsl Extension

QueryDslPredicateExecutor<Entity>

QueryDslPredicateExecutor

```
public interface StudentRepository extends CrudRepository<Student, Integer>,
                                    QueryDslPredicateExecutor {
}
```

Available methods

```
Student findOne(Predicate predicate);
Iterable<Student> findAll(Predicate predicate);
Iterable<Student> findAll(Predicate predicate, Sort sort);
Page<Student> findAll(Predicate predicate, Pageable pageable);
long count(Predicate predicate);
boolean exists(Predicate predicate);
```

Dynamic Queries

```
public class StudentExpressions {
   public static BooleanExpression hasLastName(String lastName){
       return QStudent.student.attendee.lastName.eq(lastName);
   }
   public static BooleanExpression isFullTime() {
       return QStudent.student.fullTime.eq(true);
   }
   public static BooleanExpression isOlderThan(int age){
       return QStudent.student.age.gt(age);
   }
}

studentRepository.findAll(hasLastName("Smith").and(isFullTime()).and(isOlderThan(20));
studentRepository.findAll(isFullTime().or(isOlderThan(20));
studentRepository.findAll(hasLastName("Smith").and(isOlderThan(20));
```

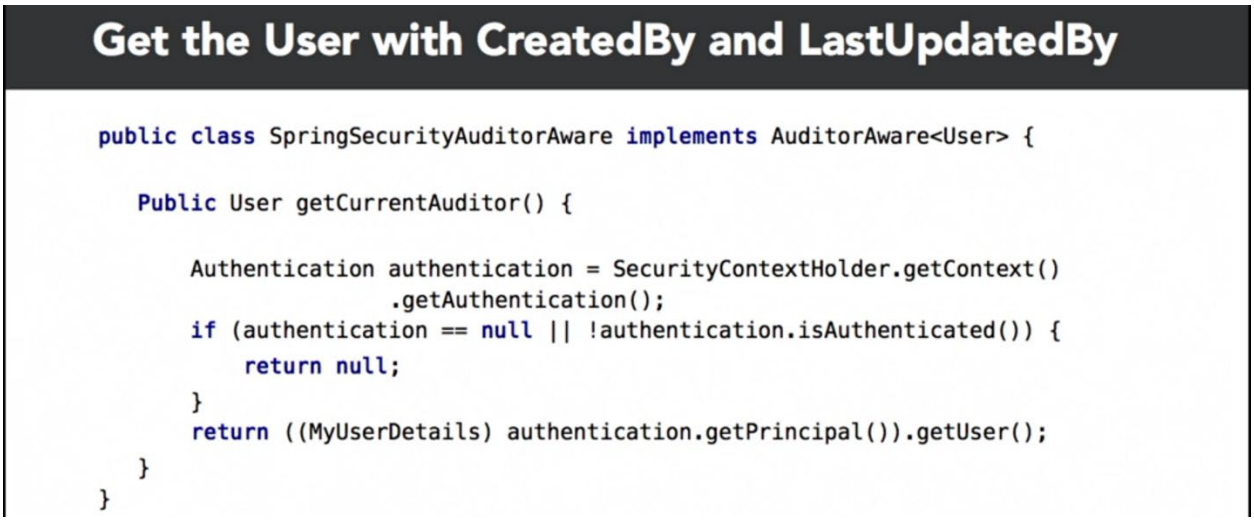
Auditing

1. **One way is to use Entity Annotations**

Entity Annotations

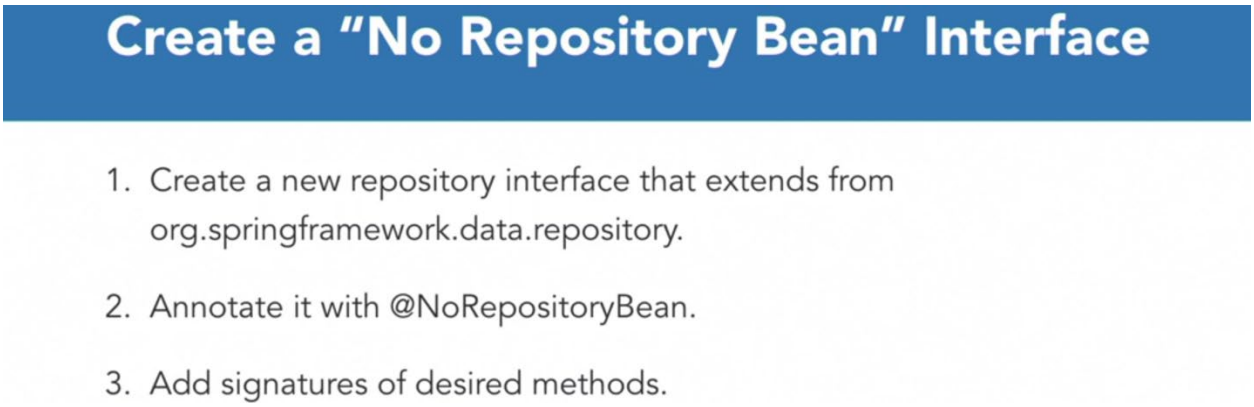
```
@CreatedDate
@Column
private ZonedDateTime createdAt;

@LastModifiedBy
@Column
private User updatedBy;

@CreatedBy
@Column
private User createdBy;
```

2. **Other way is not to touch Entity and by implementing Auditing and Extending AbstractAuditable**



Either way you need to pull the User off of the session principle and inject it into the entity .This is done by implementing AuditorAware service provider interface .

Read-only Repository



Create a ReadOnlyRepository by extending Repository interface

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


    Optional<T> findById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> iterable);

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);

    long count();

    boolean existsById(ID id);
}
```

Now extend this ReadOnlyRepositroy and add the different method signature .

```java
public interface CourseQueryRepository extends ReadOnlyRepository<Course, Integer> {
    Optional<Course> findByName(String name);

    List<Course> findByDepartmentChairMemberLastName(String chair);

    @Query("Select c from Course c where c.department.chair.member.lastName=:chair")
    List<Course> findByChairLastName(@Param("chair")String chairLastName);

    @Query("Select c from Course c join c.prerequisites p where p.id = ?1")
    List<Course> findCourseByPrerequisite(int id);

    @Query("Select new com.example.university.view.CourseView" +
            "(c.name, c.instructor.member.lastName, c.department.name) from Course c where c.id=?1")
    CourseView getCourseView(int courseId) ;

    List<Course> findByCredits(@Param("credits") int credits);

    Page<Course> findByCredits(@Param("credits") int credits, Pageable pageable);

    Course findByDepartmentName(String deptName);
```