Spring Data JPA provides repository support for the Java Persistence API (JPA).

It eases development of applications that need to access JPA data sources.

Spring Data JPA

Enhances Standard JPA with Spring

Simplifies your Data Access Layer

Repository generator

Query DSL

Auditing and Paging

Gets out of the way if needed

What is Spring Data JPA?

Spring Data JPA Dependency


```
<dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>2.1.4.RELEASE</version>
</dependency>
```

The transitive Dependencies for Spring-data-jpa:2.1.4.RELEASE

- spring-data-commons
- spring-orm
- spring-context
- spring-aop
- spring-tx
- spring-beans
- spring-core

Enabling Spring Data JPA

- **Spring Data JPA can be configured through either:**
  - "Spring Namespace" (XML configuration)
    - <jpa:repositories >
  - "Annotation-based Configuration" (Java configuration)
    - @EnableJpaRepositories

"Spring Namespace" (XML configuration)


```
<jpa:repositories
        base-package="com.pratap.db.repository"/>
```

- **Spring Namespace**
  - The JPA module of Spring Data contains a custom namespace that allows defining repository beans.
  - It also contains certain features and element attributes that are special to JPA. Generally, the JPA repositories can be set up by using the repositories element,
  - Using the repositories element looks up Spring Data repositories .
  - Beyond that, it activates persistence exception translation for all beans annotated with @Repository, to let exceptions being thrown by the JPA persistence providers be converted into Spring's DataAccessException hierarchy.

"Spring Namespace" (XML configuration)

- **Custom Namespace Attributes**
  - Beyond the default attributes of the repositories element, the JPA namespace offers additional attributes to let you gain more detailed control over the setup of the repositories:
- **entity-manager-factory-ref**
  - Explicitly wire the EntityManagerFactory to be used with the repositories being detected by the repositories element. Usually used if multiple EntityManagerFactory beans are used within the application. If not configured, Spring Data automatically looks up the EntityManagerFactory bean with the name entityManagerFactory in the ApplicationContext.
- **transaction-manager-ref**
  - Explicitly wire the PlatformTransactionManager to be used with the repositories being detected by the repositories element. Usually only necessary if multiple transaction managers or EntityManagerFactory beans have been configured. Default to a single defined PlatformTransactionManager inside the current ApplicationContext.
  - Spring Data JPA requires a PlatformTransactionManager bean named transactionManager to be present if no explicit transaction-manager-ref is defined.

Annotation-based Configuration

- The Spring Data JPA repositories support can be activated not only through an XML namespace but also by using an annotation through JavaConfig

- **@EnableJpaRepositories**

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {
 @Bean
 public DataSource dataSource() {
  EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
  return builder.setType(EmbeddedDatabaseType.HSQL).build();
 }
 @Bean
 public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

  HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
  vendorAdapter.setGenerateDdl(true);
  LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
  factory.setJpaVendorAdapter(vendorAdapter);
  factory.setPackagesToScan("com.acme.domain");
  factory.setDataSource(dataSource());
  return factory;
 }
 @Bean
 public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {

  JpaTransactionManager txManager = new JpaTransactionManager();
  txManager.setEntityManagerFactory(entityManagerFactory);
  return txManager;
 }
}
```

Boot : JPA and Spring Data JPA

- The spring-boot-starter-data-jpa POM provides a quick way to get started. It provides the following key dependencies:
  - Hibernate: One of the most popular JPA implementations.
  - Spring Data JPA: Makes it easy to implement JPA-based repositories.
  - Spring ORMs: Core ORM support from the Spring Framework.

- JPA "Entity" classes are specified in a persistence.xml file. With Spring Boot, this file is not necessary and "Entity Scanning" is used instead.
- By default, all packages below your main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) are searched.
- Any classes annotated with @Entity, @Embeddable, or @MappedSuperclass are considered.

Defining Repository Interfaces

Defining Repository Interfaces

- First, define a domain class-specific repository interface.
- The interface must extend **Repository** and be typed to the domain class and an ID type.
- If you want to expose CRUD methods for that domain type, extend **CrudRepository** instead of Repository.

Fine-tuning Repository Definition

- Typically, your repository interface extends **Repository, CrudRepository, or PagingAndSortingRepository.**

- Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with **@RepositoryDefinition**.

- Extending **CrudRepository** exposes a complete set of methods to manipulate your entities.

- If you prefer to be selective about the methods being exposed, copy the methods you want to expose from CrudRepository into your domain repository

Selectively exposing CRUD methods

- **@NoRepositoryBean**

```
interface MyBaseRepository<T, ID extends Serializable>
extends Repository<T, ID> {

        Optional<T> findById(ID id);

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

}


interface UserRepository extends
MyBaseRepository<User, Long> {

 User findByEmailAddress(EmailAddress emailAddress);

}
```

Using Repositories with Multiple Spring Data Modules

Multiple spring data module

- Using a unique Spring Data module in your application makes things simple, because all repository interfaces in the defined scope are bound to the Spring Data module.
- Sometimes, applications require using more than one Spring Data module. In such cases, a repository definition must distinguish between persistence technologies.
- When it detects multiple repository factories on the class path, Spring Data enters **strict repository configuration** mode.
- Strict configuration uses details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

Multiple spring data module

- If the repository definition extends the module-specific repository, then it is a valid candidate for the particular Spring Data module.

- If the domain class is annotated with the module-specific type annotation, then it is a valid candidate for the particular Spring Data module.

- Spring Data modules accept either third-party annotations (such as JPA's @Entity) or provide their own annotations (such as @Document for Spring Data MongoDB and Spring Data Elasticsearch).

Defining Query methods

Query methods

- The repository proxy has two ways to derive a store-specific query from the method name:
  - By deriving the query from the method name directly.
  - By using a manually defined query.

Query Lookup Strategies

- **CREATE** attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well known prefixes from the method name and parse the rest of the method.

- **USE_DECLARED_QUERY** tries to find a declared query and throws an exception if cannot find one.

- **CREATE_IF_NOT_FOUND** (default) combines CREATE and USE_DECLARED_QUERY. It looks up a declared query first, and, if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy.

Query Creation

- **Query methods**
  - Query parser will match the following:
  - find..By, query..By, read..By, count..By, get..By
  - Criteria uses JPA entity attribute names
  - Multiple criteria combined with ["And","Or"]

Query Creation

- The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository.

- The mechanism strips the prefixes **find…By, read…By, query…By, count…By, and get…By** from the method and starts parsing the rest of it.

- The introducing clause can contain further expressions, such as a Distinct to set a distinct flag on the query to be created.

- However, the first **By** acts as delimiter to indicate the start of the actual criteria.
- At a very basic level, you can define conditions on entity properties and concatenate them with And and Or.

Query DSL Overview

Query DSL

- **DSL - Domain Specific Language**
  - A domain specific language (DSL) is a customized extension of software programming language that addresses a specific business or domain.

- **Advantages of Using Query DSL**
  - Utilize your work spent on creating your JPA entities
  - Less code , less to maintain
  - Check your queries at startup rather than runtime

```
interface PersonRepository extends Repository<User, Long> {
        List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
 // Enables the distinct flag for the query
        List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
        List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);
 // Enabling ignoring case for an individual property
         List<Person> findByLastnameIgnoreCase(String lastname);
 // Enabling ignoring case for all suitable properties
        List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

 // Enabling static ORDER BY for a query
         List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
         List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

Query Method Return Type


```
public interface LocationJpaRepository extends
JpaRepository<Location,Long>{

        Location findFirstByState(String stateName);

        List<Location> findByStateLike(String
stateName);

        Long countByStateLike(String stateName);

}
```

Derived Count & Delete Query

Spring Data JPA by Pratap Kumar

- **Derived Count Query**
  - long countByLastname(String lastname);
- **Derived Delete Query**
  - long deleteByLastname(String lastname);
  - List<User> removeByLastname(String lastname);

Query Method Learning Instruction

Keyword : And and Or

Uses	Combines multiple criteria query filters together using a conditional And or Or
Keyword Example	findByStateAndCountry("CA","USA"); findByStateOrState("CA","AZ");
JPQL Example	… where a.state=?1 and a.country=?2 … where a.state=?1 or a.state=?2

Keyword : Equals, Is and Not , IsNot

Uses	The default '=' when comparing the criteria with the filter value. Use *Not* when wanting to compare not equals

Keyword Example	findByState("CA");
	findByStateIs("CA");
	findByStateEquals("CA");
	
	findByStateNot("CA");
	findByStateIsNot("CA")

JPQL Example	... where a.state=?1 ... where a.state=?1 ... where a.state=?1 ... where a.state<>?1

Keyword : Like and IsLike NotLike and IsNotLike

Uses	Useful when trying to match, or not match, a portion of the criteria filter value

Keyword Example	findByStateLike("Cali%");
	findByStateNotLIke("Al%");
	findByStateStarting

JPQL Example	… where a.state like ?1 … where a.state not like ?1

Keyword :
StartingWith,
IsStartingWith,
StartsWith,

EndingWith
IsEndingWith,
EndsWith

Containing,
IsContaining,
Contains

Uses	Similar to the "Like" keyword except the % is automatically added to the filter value

Keyword Example	findByStateStartingWith("Al"); // Al%
	findByStateEndingWith("ia");// %ia
	findByStateContaining("in");// %in%

JPQL Example	...where a.state like ?1 ...where a.state like ?1 ...where a.state like ?1

Keyword : LessThan(Equal) and GreaterThan(Equal)

Uses	when you need to perform a< , <= , > , >= comparision with number data types
Keyword Example	findByPriceLessThan(20); findByPriceLessThanEqual(20) findByPriceGreaterThan(20) findByPriceGreaterThanEqual(20)
JPQL Example	...where a.price <= ?1 ...where a.price <= ?1 ...where a.price > ?1 ...where a.price >= ?1

Keyword : Before, After and Between

Uses	When you need to perform a less than, greater then or range comparison with date/time data types

Keyword Example	findByFoundedDateBefore(dateObj);
	findByFoundedDateAfter(dateObj);
	findByFoundedDateBetween(startDate , endDate);

JPQL Example	...where a.foundedDate < ?1 ...where a.foundedDate > ?1 ...where a.foundedDate between ?1 and ?2

Keyword : True and False

Uses	Useful when comparing boolean values with true or false
Keyword Example	findByActiveTrue(); findByActiveFalse();
JPQL Example	...where a.active = true ...where a.active = false

Keyword : IsNull, IsNotNull andNotNull

Uses	Used to check whether a criteria value is null or not null

Keyword Example	findByStateIsNull();
	findByStateIsNotNull();
	findByStateNotNull();

JPQL Example	...where a.state is null ...where a.state not null ...where a.state not null

Keyword : In and NotIn

Uses	When you need to test if a column value is aprt of a collection or set of values of not
Keyword Example	findByStateIn(Collection<String> states); findByStateNotIn(Collection<String> states)
JPQL Example	...where a.state in ?1 ...where a.state not in ?1

Keyword : IgnoreCase

Uses	When you need to perform a case insensitiv comparison
Keyword Example	findByStateIgnoreCase("ca"); findByStateStartingWithIgnoreCase("c");
JPQL Example	...where upper(a.state) = uper(?1) ...where upper(a.state) like upper(?1%)

Keyword : OrderBy

Uses	Used to setup an order by clause on your query
Keyword Example	findByStateOrderByCountryAsc(); findByStateOrderByCountryDesc();
JPQL Example	… where a.state order by a.country asc; … where a.state order by a.country desc;

Keyword : First,Top and Distinct

Uses	used to limit the results returned by the query

Keyword Example	findFirstByStateLike("Al");
	findTop5ByStateLike("A");
	findDistinctManufacturerByStateLike("A");

JPQL Example	...where a.state like ?1 limit 1 ...where a.state like ?1 limit 5 select distinct ... where a.state like ?1

More Fun with Queries

@Query Annotation

- Reuse existing JPQL in your data access layer
- Advanced query functionality
- Eager loading control ("fetch")

- // JPQL (Indexed Parameter)

```
@Query("select e from Emp e where e.ename=?1");
List<Emp> qeuryByEmployeeName(String name);
```

- //JPQL (Named Parameter)

```
@Query("select e from Emp e where e.deptno=:deptno");
List<Emp> queryEmployeeByDeptno(@Param("deptno") String dno);
```

@Query Annotation

- Reuse existing JPQL in your data access layer
- Advanced query functionality
- Eager loading control ("fetch")

@Query Options

- **Named Parameter**
  - @Query("select e from Emp m where m.ename = :ename");
  - List<Emp> queryByName(@Param("ename") String name);
- **Enhanced JPQL Syntax**
  - @Query("select e from Emp e where e.ename like %?1")
  - List<Emp> queryByName(String name);
- **Native Queries**
  - @Query(value="select * from Emp where name=?1", **nativeQuery=true**)
  - List<Emp> queryByName(String name)
- **Modifiable Queries**
  - @Modifying
  - @Query("update Emp e set e.ename=?1")
  - int updateByName(String name);

JPA Named Queries

- App startup validates queries rather than runtime

```
@Entity
@NamedQuery(
        name="Emp.namedFindAllEmpByDeptno" ,
        query="select m from Emp where e.deptno=:deptno")
public class Emp{ … }
```

```
@Repository
public interface EmpJpaRepository extends JpaRepository<Emp,Long >{
        List<Emp> namedFindAllEmpByDeptno(@Param("deptno") String deptno);
}
```

- or

```
@Query(name="Emp.namedFindAllEmpByDeptno")
List<Emp> findAllEmpByDeptno(@Param("deptno") String deptno);
```

Query Precedence

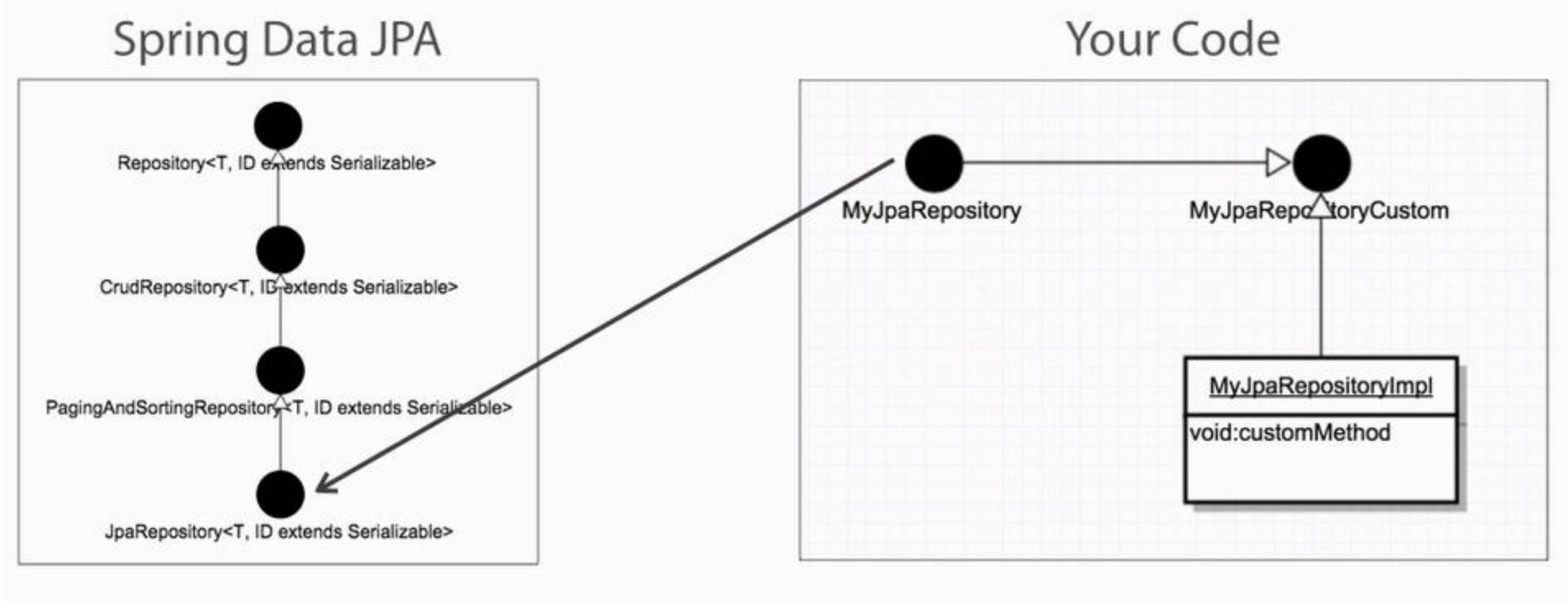
- **JpaRepository Query Precedence**
  - Methods with @Query annotation take highest precedence
  - Methods that match a named or native named query name
  - Methods that follow the query DSL keyword naming structure

Advanced Features

Paging & Sorting

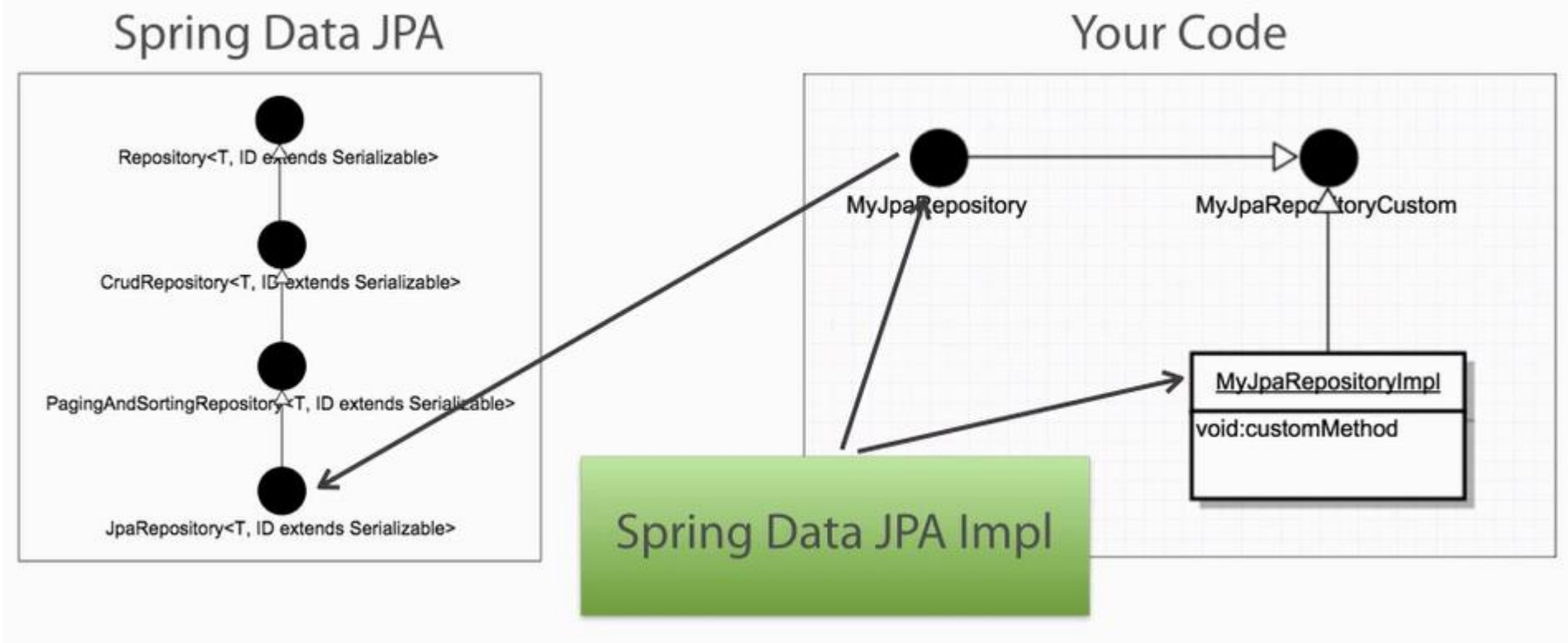
Spring Data JPA by Pratap Kumar

- PagingAndSortingRepository
  - Page<T> findAll(Pageable pageable)
  - Iterable<T> findAll(Sort sort)
- Pageable
- Page
- Sort




```
<jpa:repositories   base-package="com.pratap.repository"/>
<jpa:repositories   base-package="com.pratap.repository"
                    repository-impl-postfix="CustomClass"/>
```

Custom Repositories



Custom Repositories

Auditing Support

- Spring Data JPA Annotations
  - @CreatedBy
  - @CreatedDate
  - @LastModifiedBy
  - @LastModifiedDate

```
@Entity
public class Model {
  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;
}
```

```
<jpa:auditing auditor-aware-ref="securityAuditorAware" />
```

```
@Configuration
@EnableJpaAuditing
public class Config {
  @Bean
  public AuditorAware<User> auditorProvider() {
    return new SecurityAuditorAware();
  }
}
```

- Spring Data JPA Annotations
  - @CreatedBy
  - @CreatedDate
  - @LastModifiedBy
  - @LastModifiedDate

```
public class SecurityAuditorAware
          implements AuditorAware<User> {

  public User getCurrentAuditor() {
   //...
    return user;
  }
}
```

Auditing Support

Locking

Spring Data JPA by Pratap Kumar