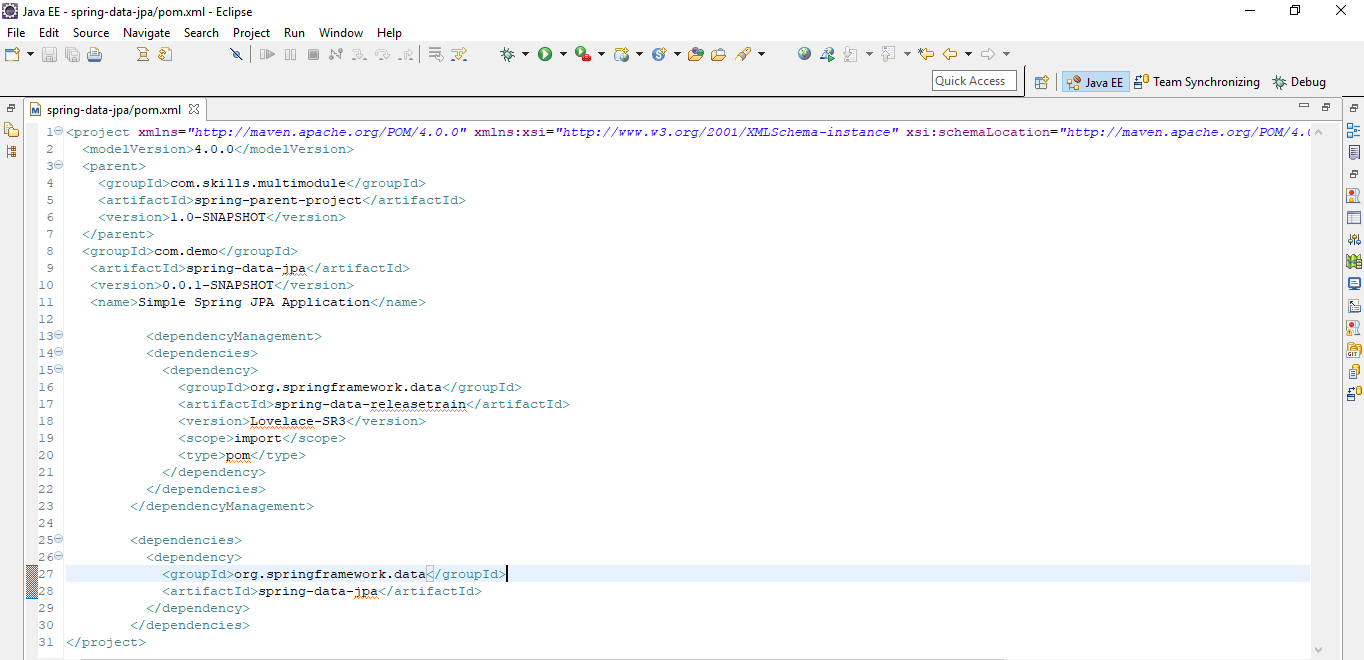
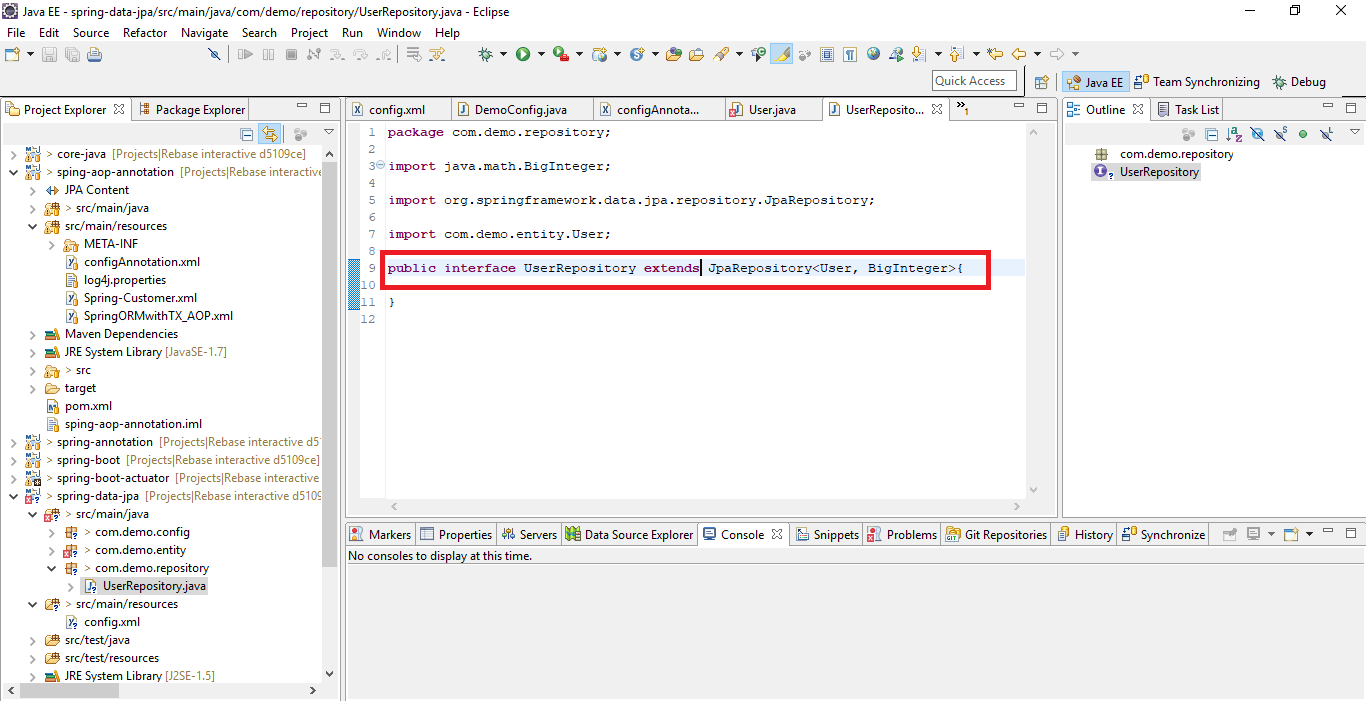
<https://docs.spring.io/spring-data/jpa/docs/2.1.3.RELEASE/reference/html/>

**Dependencies**



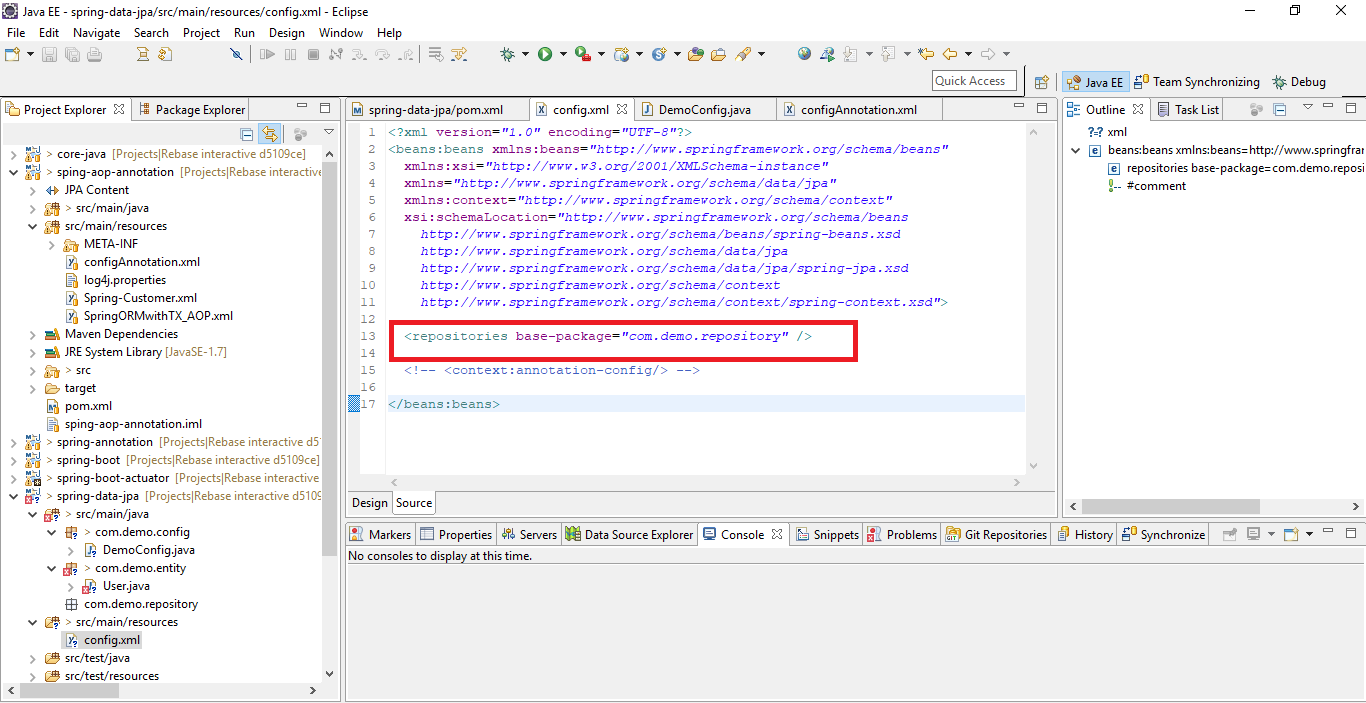
**Repository**

The central interface in the Spring Data repository abstraction is Repository. It takes the domain class to manage as well as the ID type of the domain class as type arguments

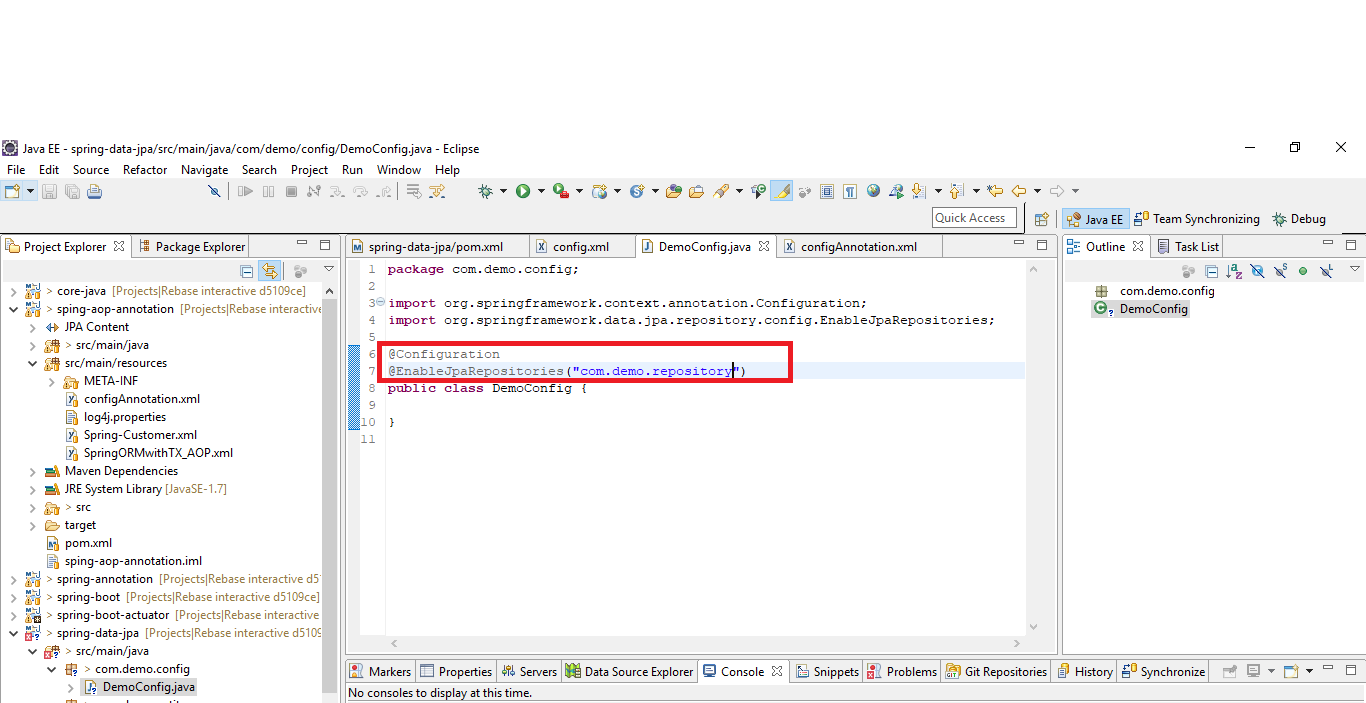


**To enable repositories and define base packages to scan for repositories**

**XML**

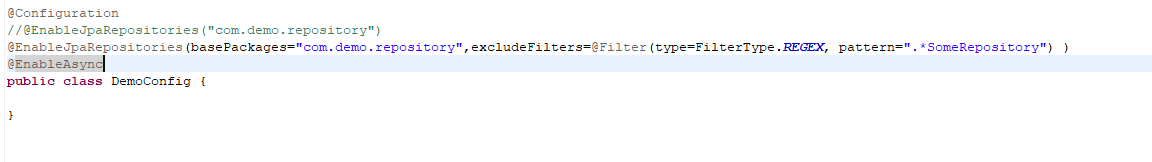


Annotation

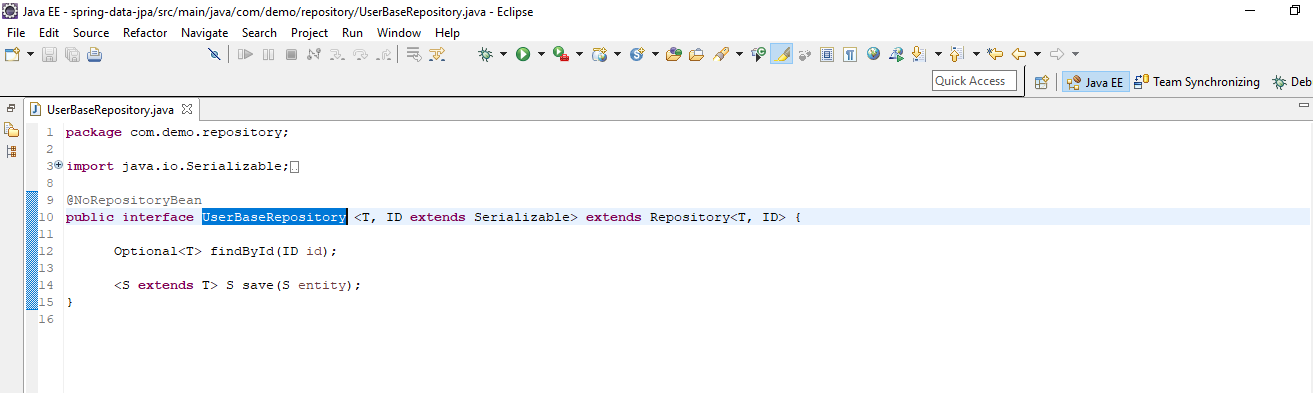


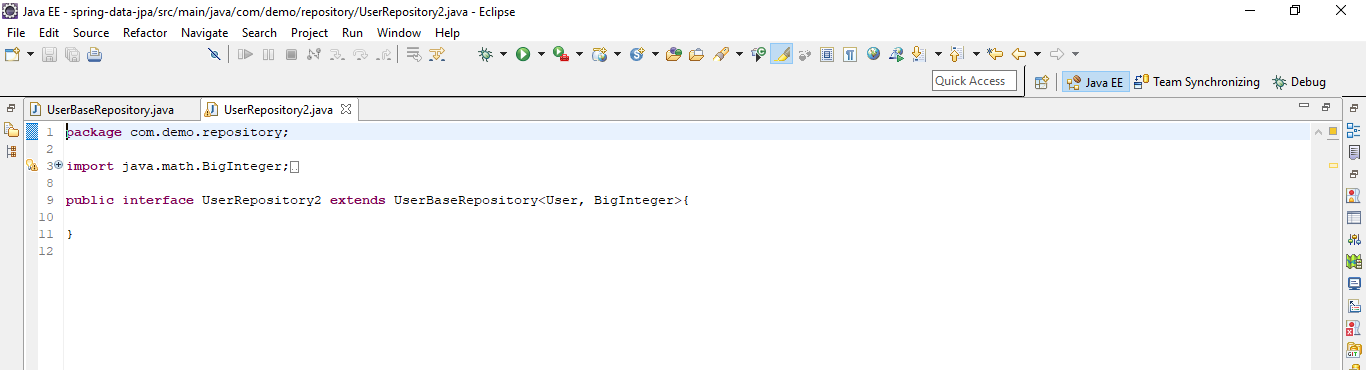
Using filters

The below example excludes all interfaces ending in SomeRepository from being instantiated.



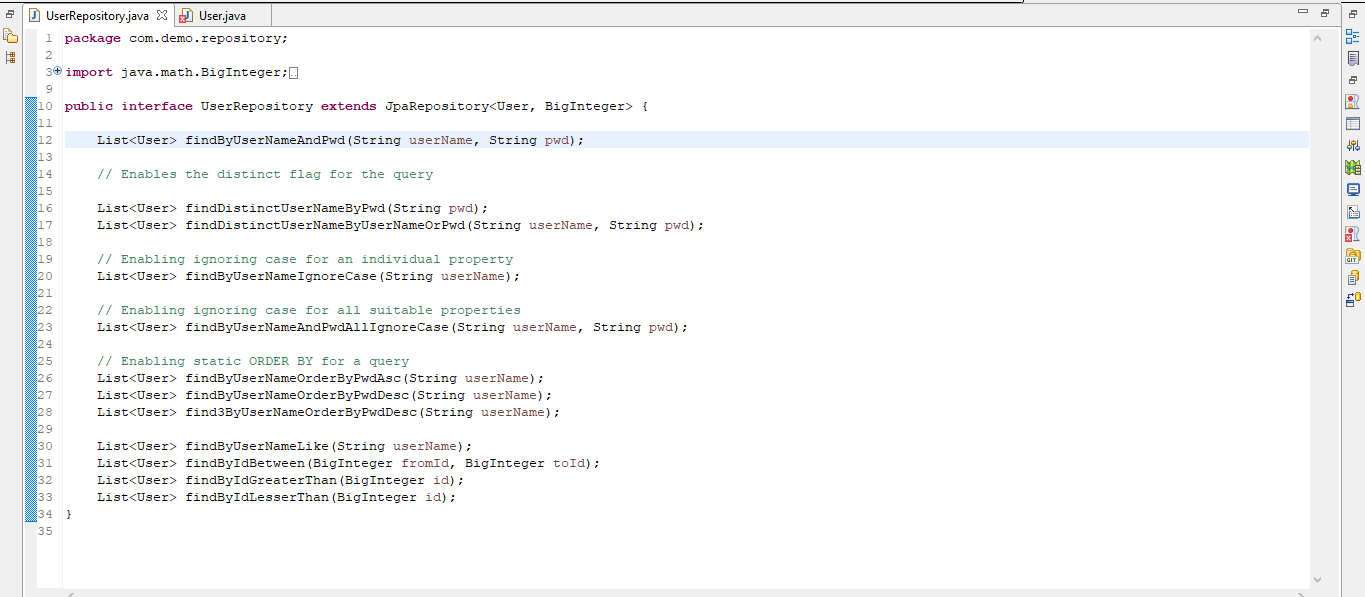
If we want we can include selective methods from repository interface





**Query Creation**

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. The following example shows how to create a number of queries:



| *able 3. Supported keywords inside method names* | | |
| --- | --- | --- |
| **Keyword** | **Sample** | **JPQL snippet** |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … 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 |
| 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> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

**PropertyExpressions**

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time, you already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. Consider the following method signature:

List<Person> findByAddressZipCode(ZipCode zipCode);

Assume a Person has an Address with a ZipCode. In that case, the method creates the property traversal x.address.zipCode. The resolution algorithm starts by interpreting the entire part (AddressZipCode) as the property and checks the domain class for a property with that name (uncapitalized). If the algorithm succeeds, it uses that property. If not, the algorithm splits up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property — in our example, AddressZip and Code. If the algorithm finds a property with that head, it takes the tail and continues building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm moves the split point to the left (Address, ZipCode) and continues.

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the Person class has an addressZip property as well. The algorithm would match in the first split round already, choose the wrong property, and fail (as the type of addressZip probably has no code property).

To resolve this ambiguity you can use \_ inside your method name to manually define traversal points. So our method name would be as follows:

List<Person> findByAddress\_ZipCode(ZipCode zipCode);

Because we treat the underscore character as a reserved character, we strongly advise following standard Java naming conventions (that is, not using underscores in property names but using camel case instead).

Ref - <http://www.thejavageek.com/2017/02/25/spring-data-jpa-query-methods/>

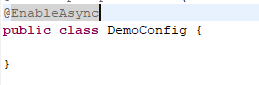
**Limiting Query Results**

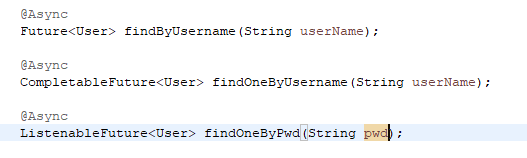
The results of query methods can be limited by using the first or top keywords, which can be used interchangeably. An optional numeric value can be appended to top or first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:



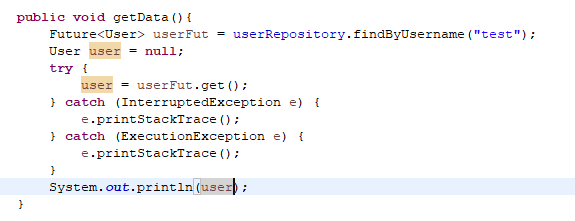
**Async query results**

Enable async

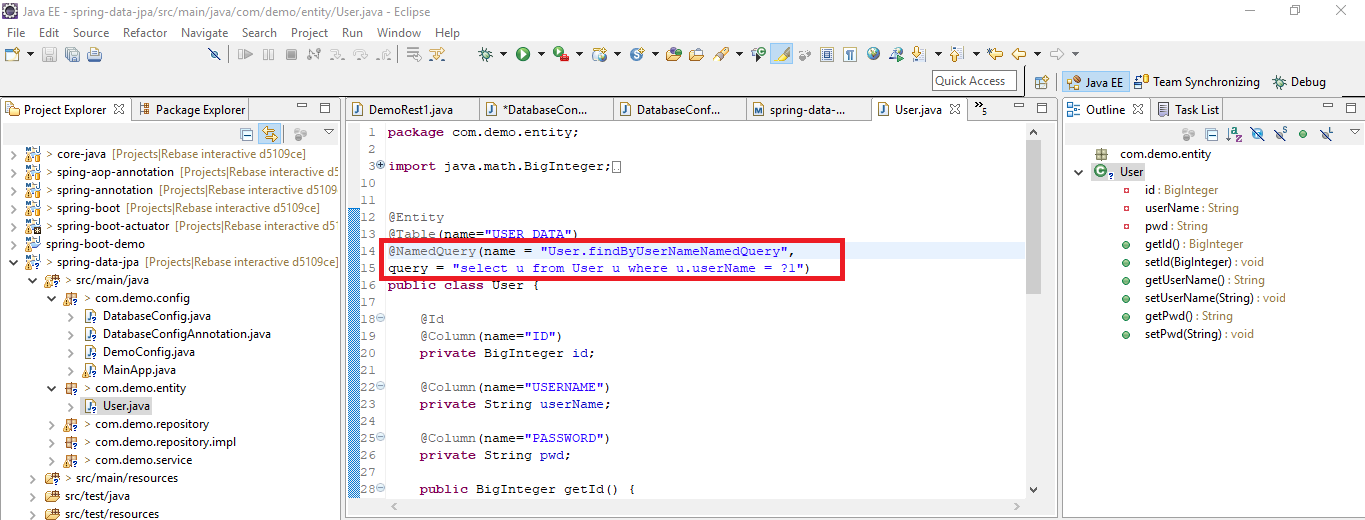




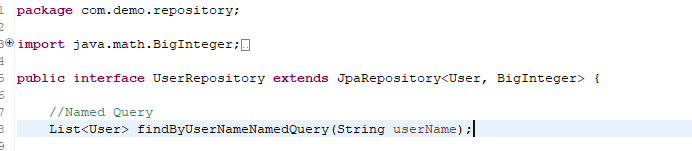
Usage



**@NamedQuery**

****

Use in repository



Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined in the examlpe instead of trying to create a query from the method name.

**@Query**

Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that executes them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

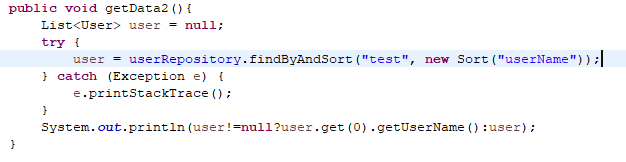
**Native Query**

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example

**Sort**

Sorting can be done be either providing a PageRequest or by using Sort directly. The properties actually used within the Order instances of Sort need to match your domain model, which means they need to resolve to either a property or an alias used within the query



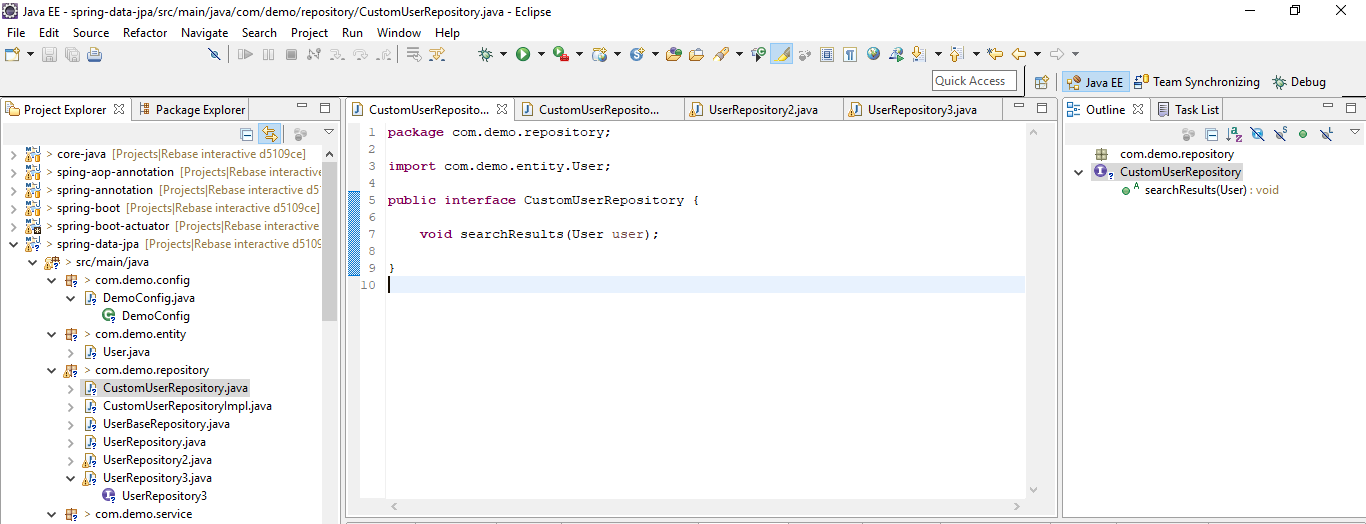


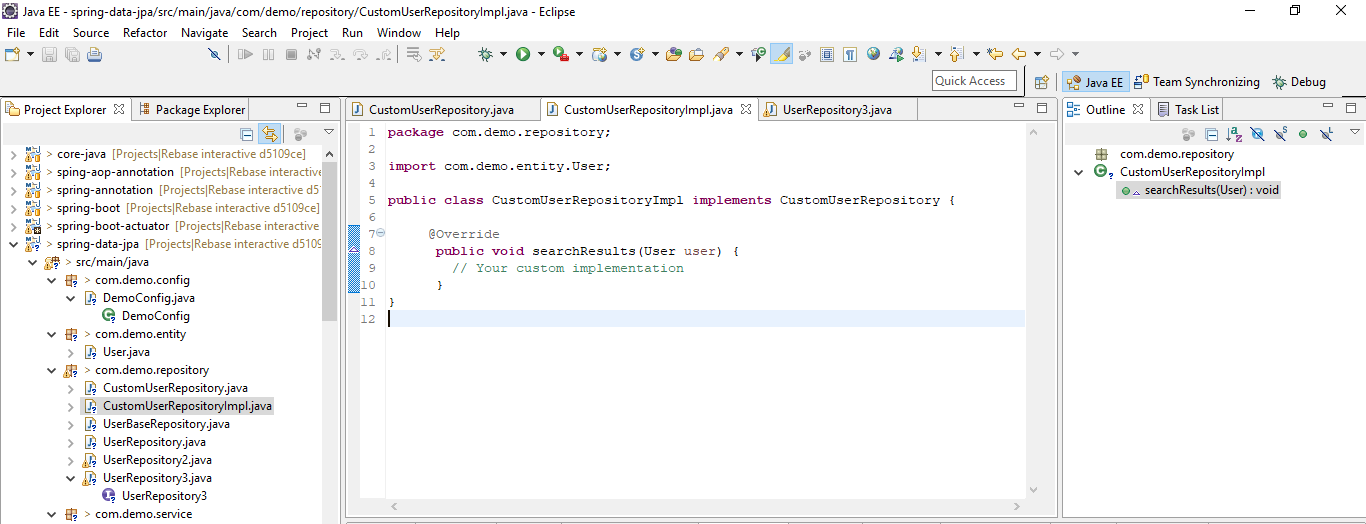
**Named Parameters**

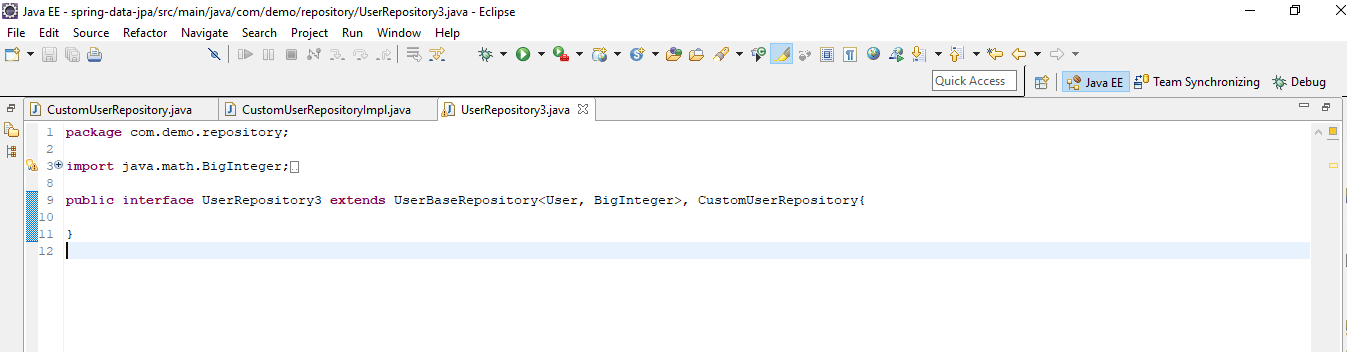
By default, Spring Data JPA uses position-based parameter binding, as described in all the preceding examples. This makes query methods a little error-prone when refactoring regarding the parameter position. To solve this issue, you can use @Param annotation to give a method parameter a concrete name and bind the name in the query, as shown in the following example:



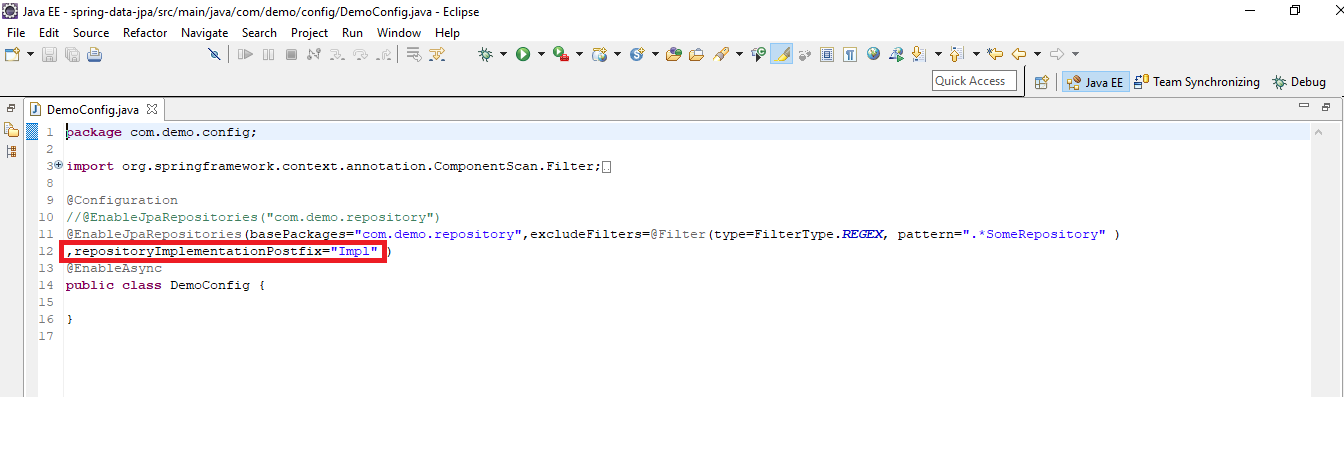
**Custom Repository**



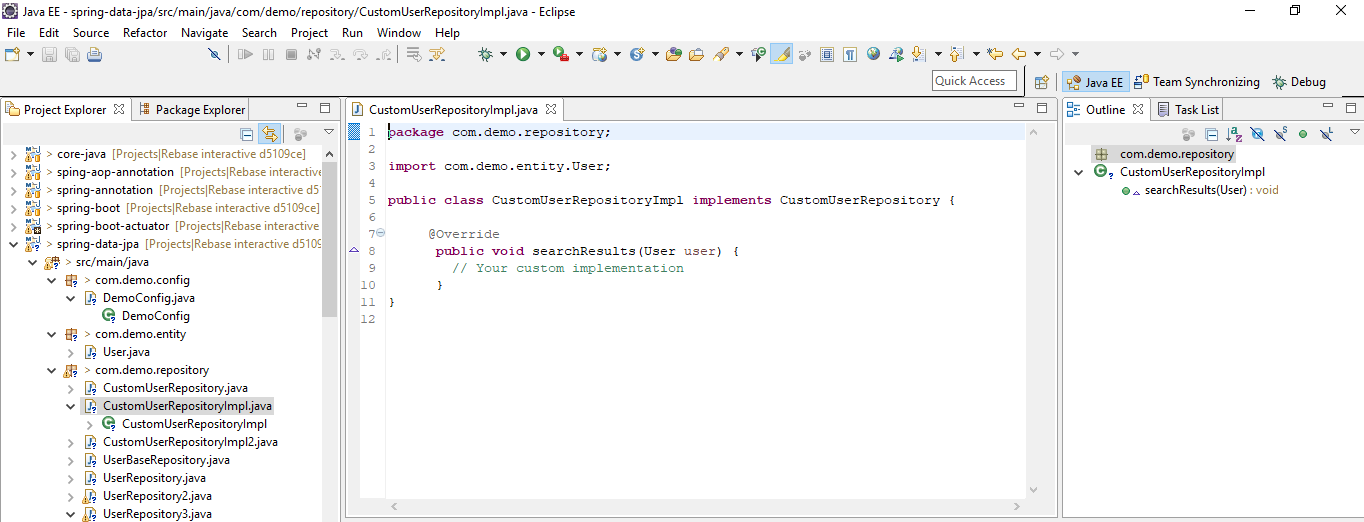


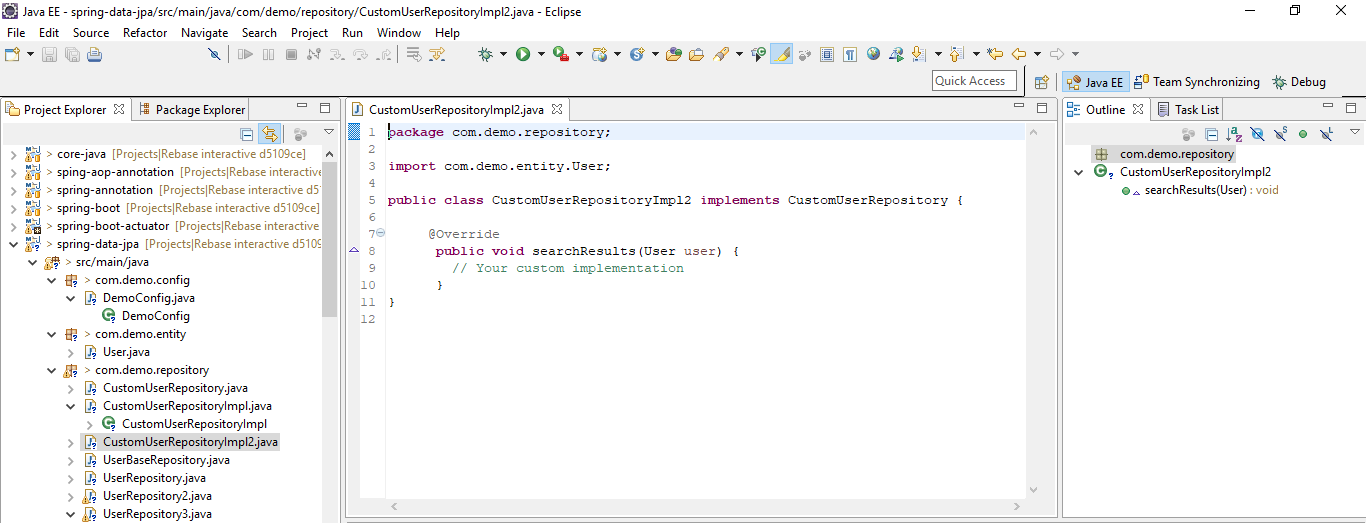


If you use namespace configuration, the repository infrastructure tries to autodetect custom implementation fragments by scanning for classes below the package in which it found a repository. These classes need to follow the naming convention of appending the namespace element’s repository-impl-postfix attribute to the fragment interface name. This postfix defaults to Impl. The following example shows a repository that uses the default postfix and a repository that sets a custom value for the postfix:

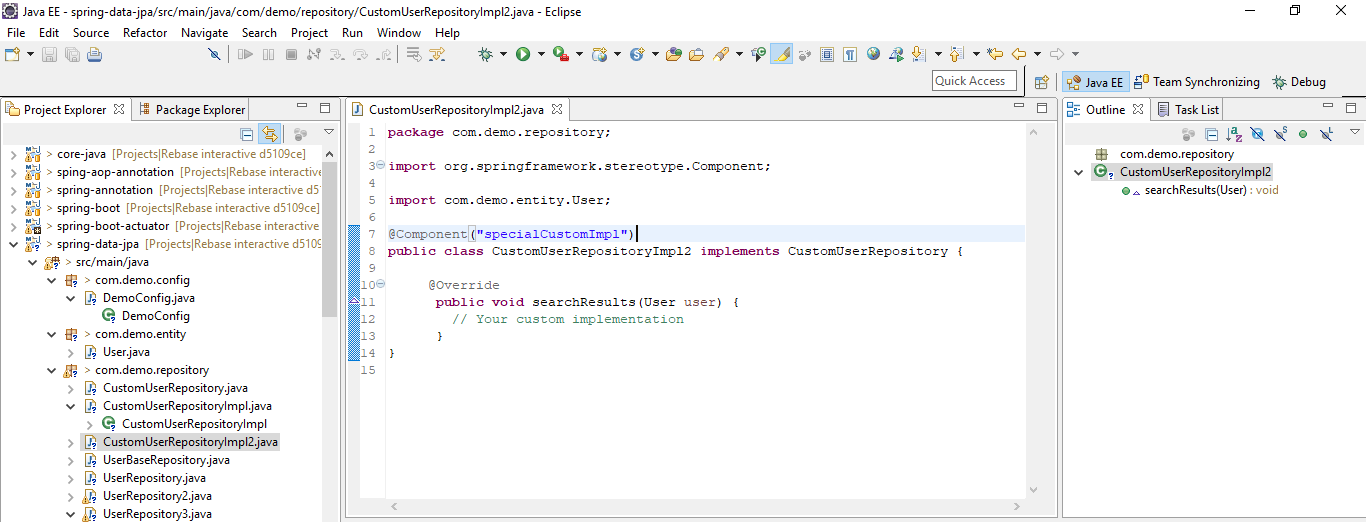


Resolving ambiguity like below





Here we must resolve by giving bean name



**@DynamicInsert**

The dynamic-insert attribute tells Hibernate whether to include null properties in the SQL INSERT statement.

**@DynamicUpdate**

The dynamic-update attribute tells Hibernate whether to include unmodified properties in the SQL UPDATE statement.

**QueryDSL**

Querydsl is a framework that enables the construction of statically typed SQL-like queries through its fluent API.

Several Spring Data modules offer integration with Querydsl through QuerydslPredicateExecutor, as shown in the following example:

public interface QuerydslPredicateExecutor<T> {

Optional<T> findById(Predicate predicate);

Iterable<T> findAll(Predicate predicate);

long count(Predicate predicate);

boolean exists(Predicate predicate);

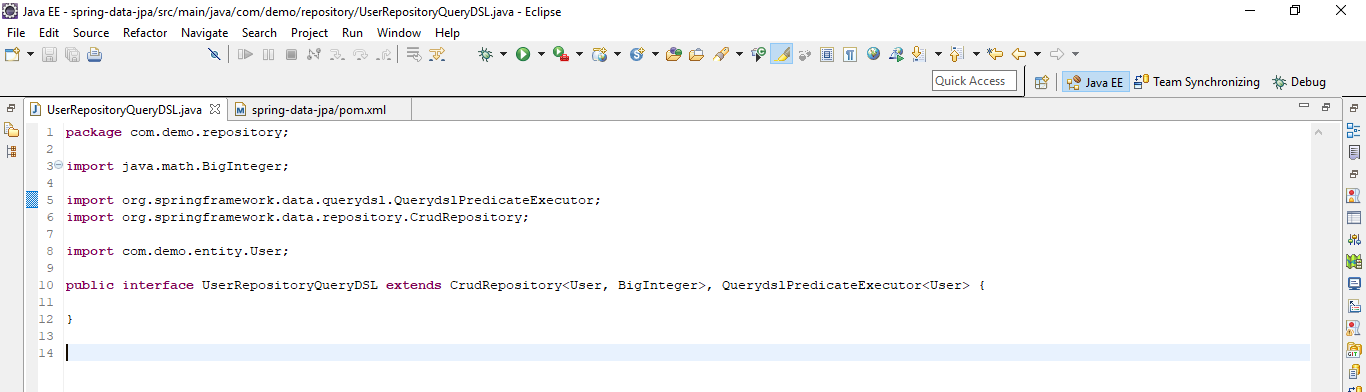
// … more functionality omitted.

}

Dependencies

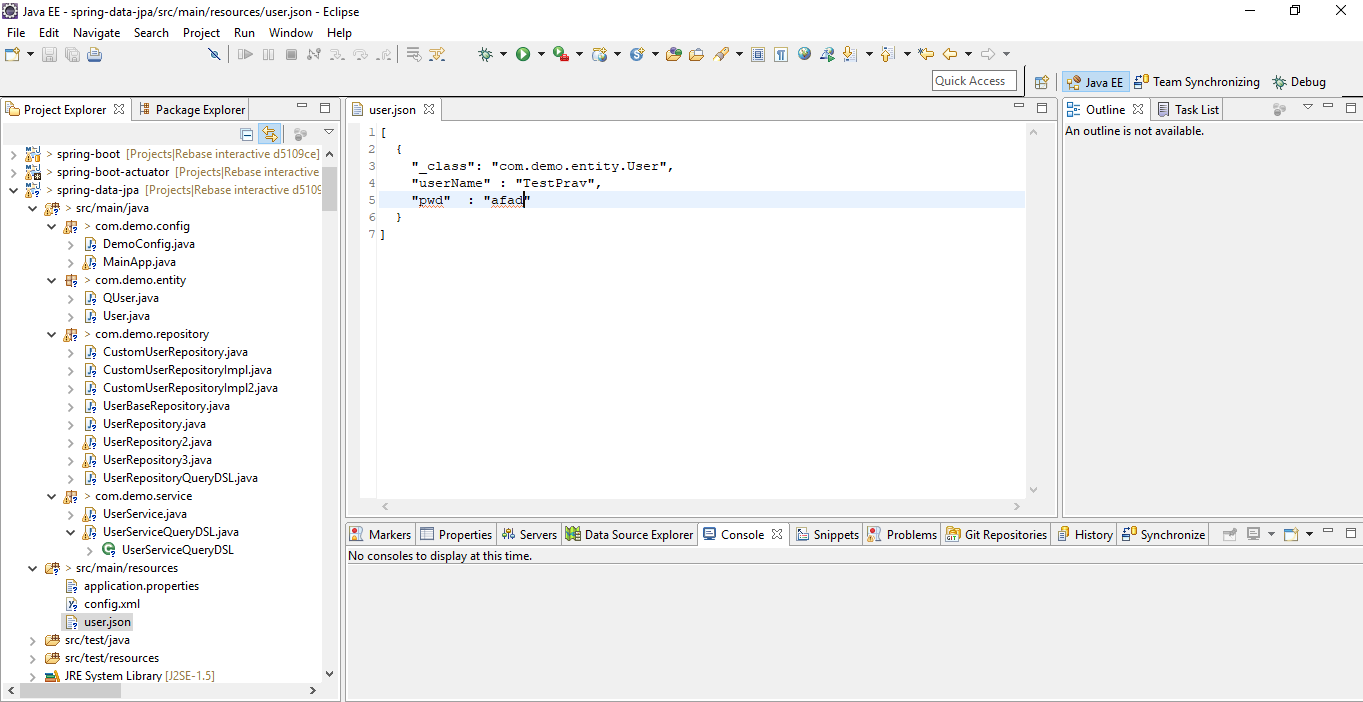


Interface

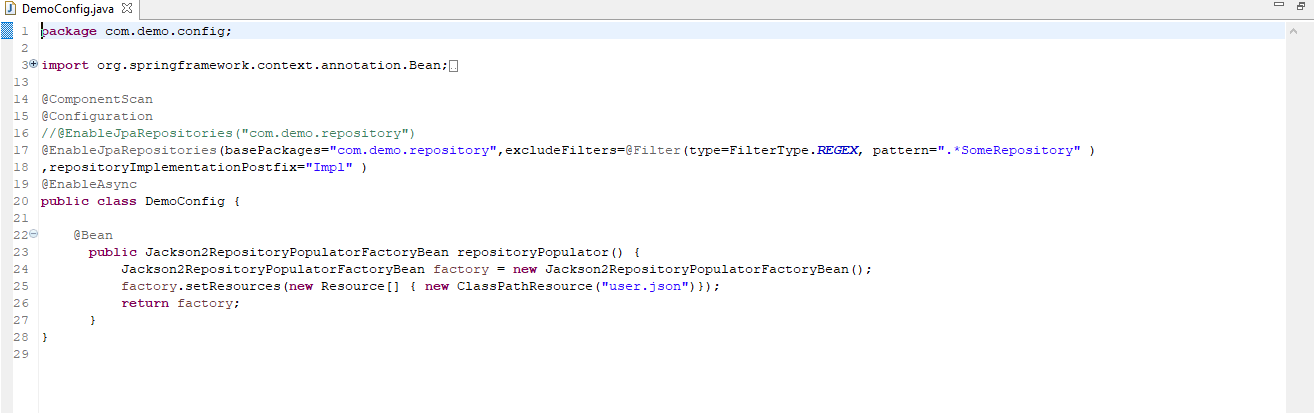


Get data from JSon

Create Json



Refer that json



Run and verify, we get data from json

**Bootstrap JPA entity**

As of Spring Data JPA 2.1 you can now configure a BootstrapMode (either via the @EnableJpaRepositories annotation or the XML namespace) that takes the following values:

DEFAULT (default) — Repositories are instantiated eagerly unless explicitly annotated with @Lazy. The lazification only has effect if no client bean needs an instance of the repository as that will require the initialization of the repository bean.

LAZY — Implicitly declares all repository beans lazy and also causes lazy initialization proxies to be created to be injected into client beans. That means, that repositories will not get instantiated if the client bean is simply storing the instance in a field and not making use of the repository during initialization. Repository instances will be initialized and verified upon first interaction with the repository.

DEFERRED — Fundamentally the same mode of operation as LAZY, but triggering repository initialization in response to an ContextRefreshedEvent so that repositories are verified before the application has completely started.

**Saving Entities**

Saving an entity can be performed with the CrudRepository.save(…) method. It persists or merges the given entity by using the underlying JPA EntityManager. If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the entityManager.persist(…) method. Otherwise, it calls the entityManager.merge(…) method.

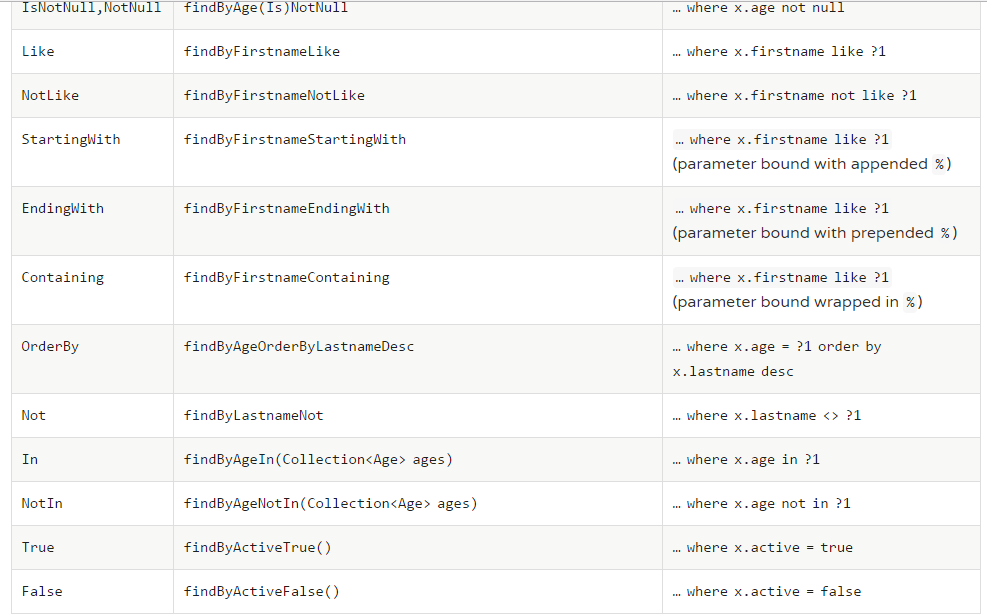
Entity State-detection Strategies

Spring Data JPA offers the following strategies to detect whether an entity is new or not:

Id-Property inspection (default): By default Spring Data JPA inspects the identifier property of the given entity. If the identifier property is null, then the entity is assumed to be new. Otherwise, it is assumed to be not new. If Id-property has value, check if it exists then merge else persist

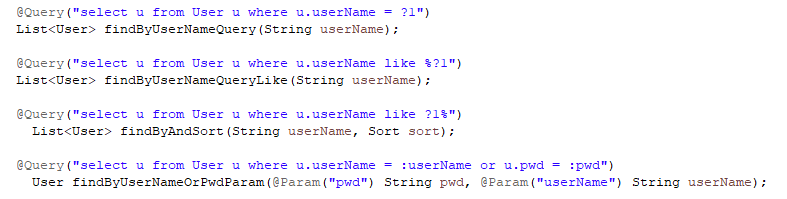
**Query Methods**







@Query



Native Query



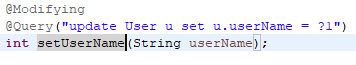
Using SpEL

As of Spring Data JPA release 1.4, we support the usage of restricted SpEL template expressions in manually defined queries that are defined with @Query. Upon query execution, these expressions are evaluated against a predefined set of variables. Spring Data JPA supports a variable called entityName. Its usage is select x from #{#entityName} x. It inserts the entityName of the domain type associated with the given repository. The entityName is resolved as follows: If the domain type has set the name property on the @Entity annotation, it is used. Otherwise, the simple class-name of the domain type is used.



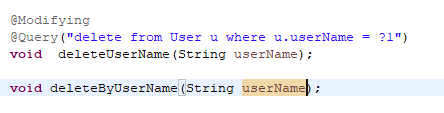
In the preceding example, the MappedTypeRepository interface is the common parent interface for a few domain types extending AbstractMappedType. It also defines the generic findAllByAttribute(…) method, which can be used on instances of the specialized repository interfaces. If you now invoke findByAllAttribute(…) on ConcreteRepository, the query becomes select t from ConcreteType t where t.attribute = ?1.

Modifying Queries





Doing so triggers the query annotated to the method as an updating query instead of a selecting one. If you wish the EntityManager to be cleared automatically, you can set the @Modifying annotation’s clearAutomatically attribute to true.



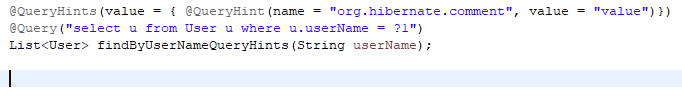
Important difference between these two is

the latter method issues a single JPQL query (the one defined in the annotation) against the database. This means even currently loaded instances of User do not see lifecycle callbacks invoked.

To make sure lifecycle queries are actually invoked, an invocation of deleteByUserName(…) executes a query and then deletes the returned instances one by one, so that the persistence provider can actually invoke @PreRemove callbacks on those entities.

Query Hints

To apply JPA query hints to the queries declared in your repository interface, you can use the @QueryHints annotation. It takes an array of JPA @QueryHint annotations



Fetch and LoadGraphs

The JPA 2.1 specification introduced support for specifying Fetch- and LoadGraphs that we also support with the @EntityGraphannotation, which lets you reference a @NamedEntityGraph definition. You can use that annotation on an entity to configure the fetch plan of the resulting query. The type (Fetch or Load) of the fetching can be configured by using the type attribute on the @EntityGraph annotation. See the JPA 2.1 Spec 3.7.4 for further reference.

The definition of an entity graph is independent of the query and defines which attributes to fetch from the database. An entity graph can be used as a fetch or a load graph. If a fetch graph is used, only the attributes specified by the entity graph will be treated as FetchType.EAGER. All other attributes will be lazy. If a load graph is used, all attributes that are not specified by the entity graph will keep their default fetch type.

The following example shows how to define a named entity graph on an entity:

*Example 67. Defining a named entity graph on an entity.*

@Entity

@NamedEntityGraph(name = "GroupInfo.detail",

attributeNodes = @NamedAttributeNode("members"))

public class GroupInfo {

// default fetch mode is lazy.

@ManyToMany

List<GroupMember> members = new ArrayList<GroupMember>();

…

}

The following example shows how to reference a named entity graph on a repository query method:

*Example 68. Referencing a named entity graph definition on a repository query method.*

@Repository

public interface GroupRepository extends CrudRepository<GroupInfo, String> {

@EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD)

GroupInfo getByGroupName(String name);

}

It is also possible to define ad hoc entity graphs by using @EntityGraph. The provided attributePaths are translated into the according EntityGraph without needing to explicitly add @NamedEntityGraph to your domain types, as shown in the following example:

*Example 69. Using AD-HOC entity graph definition on an repository query method.*

@Repository

public interface GroupRepository extends CrudRepository<GroupInfo, String> {

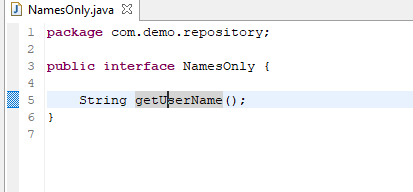
@EntityGraph(attributePaths = { "members" })

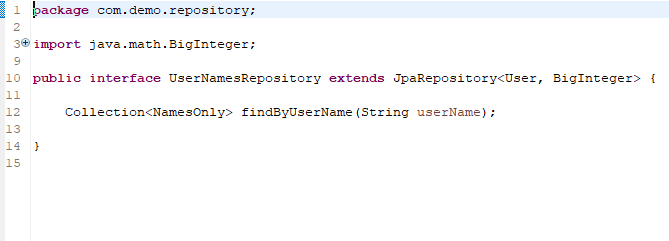
GroupInfo getByGroupName(String name);

}

Projections

Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository. However, it might sometimes be desirable to create projections based on certain attributes of those types. Spring Data allows modeling dedicated return types, to more selectively retrieve partial views of the managed aggregates.





**Stored Procedure**

The JPA 2.1 specification introduced support for calling stored procedures by using the JPA criteria query API. We Introduced the @Procedure annotation for declaring stored procedure metadata on a repository method.

The examples to follow use the following procedure:

*Example 82. The definition of the plus1inout procedure in HSQL DB.*

/;

DROP procedure IF EXISTS plus1inout

/;

CREATE procedure plus1inout (IN arg int, OUT res int)

BEGIN ATOMIC

set res = arg + 1;

END

/;

Metadata for stored procedures can be configured by using the NamedStoredProcedureQuery annotation on an entity type.

*Example 83. StoredProcedure metadata definitions on an entity.*

@Entity

@NamedStoredProcedureQuery(name = "User.plus1", procedureName = "plus1inout", parameters = {

@StoredProcedureParameter(mode = ParameterMode.IN, name = "arg", type = Integer.class),

@StoredProcedureParameter(mode = ParameterMode.OUT, name = "res", type = Integer.class) })

public class User {}

You can reference stored procedures from a repository method in multiple ways. The stored procedure to be called can either be defined directly by using the value or procedureName attribute of the @Procedure annotation or indirectly by using the nameattribute. If no name is configured, the name of the repository method is used as a fallback.

The following example shows how to reference an explicitly mapped procedure:

*Example 84. Referencing explicitly mapped procedure with name "plus1inout" in database.*

@Procedure("plus1inout")

Integer explicitlyNamedPlus1inout(Integer arg);

The following example shows how to reference an implicitly mapped procedure by using a procedureName alias:

*Example 85. Referencing implicitly mapped procedure with name "plus1inout" in database via procedureName alias.*

@Procedure(procedureName = "plus1inout")

Integer plus1inout(Integer arg);

The following example shows how to reference an explicitly mapped named procedure in EntityManager:

*Example 86. Referencing explicitly mapped named stored procedure "User.plus1IO" in EntityManager.*

@Procedure(name = "User.plus1IO")

Integer entityAnnotatedCustomNamedProcedurePlus1IO(@Param("arg") Integer arg);

The following example shows how to reference an implicitly named stored procedure in EntityManager by using the method name:

*Example 87. Referencing implicitly mapped named stored procedure "User.plus1" in EntityManager by using the method name.*

@Procedure

Integer plus1(@Param("arg") Integer arg);

**Query By**

The Query by Example API consists of three parts:

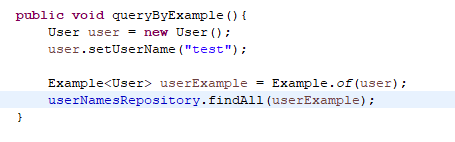
* Probe: The actual example of a domain object with populated fields.
* ExampleMatcher: The ExampleMatcher carries details on how to match particular fields. It can be reused across multiple Examples.
* Example: An Example consists of the probe and the ExampleMatcher. It is used to create the query.

Query by Example is well suited for several use cases:

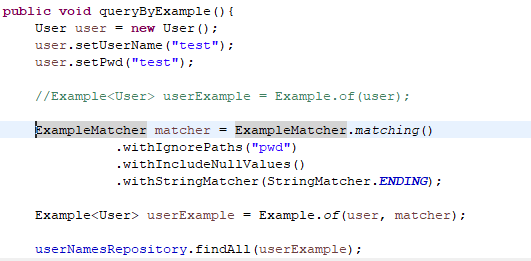
* Querying your data store with a set of static or dynamic constraints.
* Frequent refactoring of the domain objects without worrying about breaking existing queries.
* Working independently from the underlying data store API.

Query by Example also has several limitations:

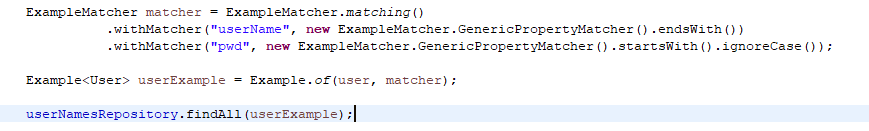
* No support for nested or grouped property constraints, such as firstname = ?0 or (firstname = ?1 and lastname = ?2).
* Only supports starts/contains/ends/regex matching for strings and exact matching for other property types.

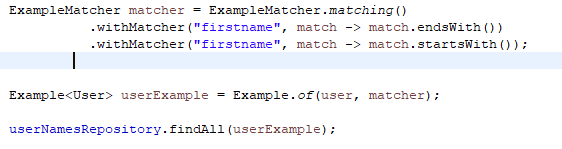


Example Matcher



By default, the ExampleMatcher expects all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use ExampleMatcher.matchingAny().





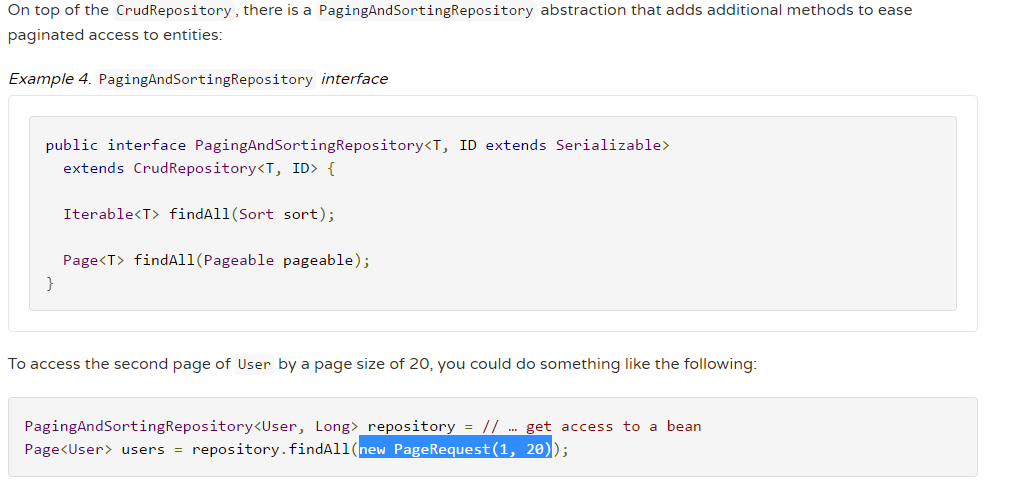
ExampleMatcher are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings. The following table describes the scope of the various ExampleMatcher settings:

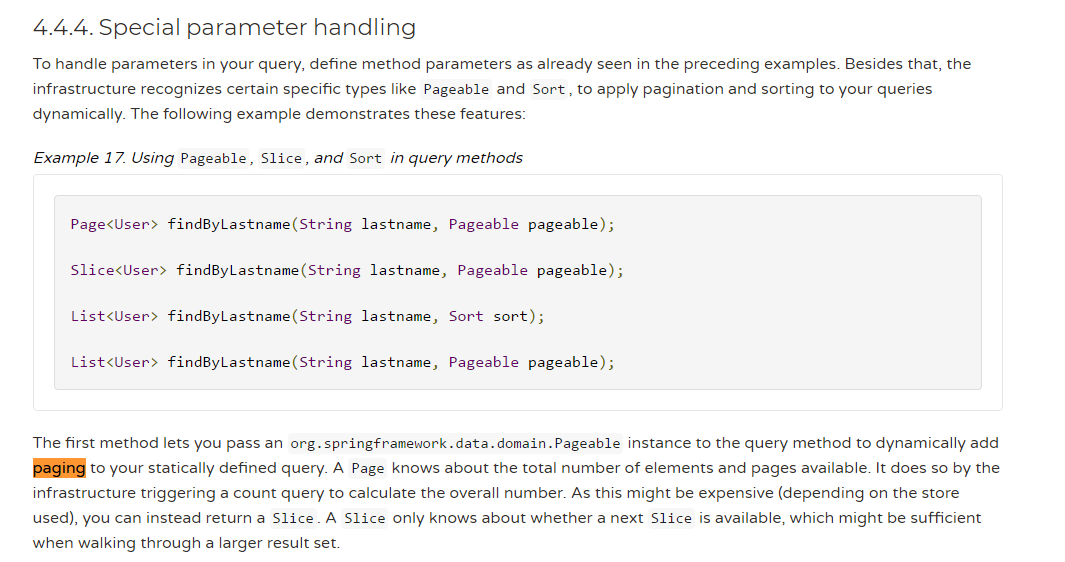
| *Table 4. Scope of ExampleMatcher settings* | |
| --- | --- |
| **Setting** | **Scope** |
| Null-handling | ExampleMatcher |
| String matching | ExampleMatcher and property path |
| Ignoring properties | Property path |
| Case sensitivity | ExampleMatcher and property path |
| Value transformation | Property path |

The following table shows the various StringMatcher options that you can use and the result of using them on a field named firstname:

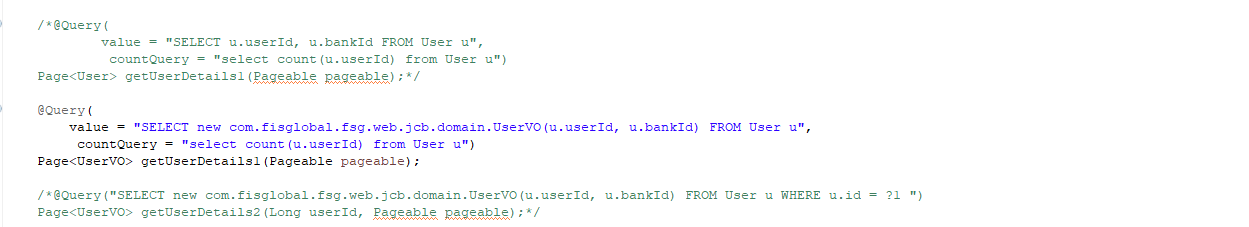
| *Table 5. StringMatcher options* | |
| --- | --- |
| **Matching** | **Logical result** |
| DEFAULT (case-sensitive) | firstname = ?0 |
| DEFAULT (case-insensitive) | LOWER(firstname) = LOWER(?0) |
| EXACT (case-sensitive) | firstname = ?0 |
| EXACT (case-insensitive) | LOWER(firstname) = LOWER(?0) |
| STARTING (case-sensitive) | firstname like ?0 + '%' |
| STARTING (case-insensitive) | LOWER(firstname) like LOWER(?0) + '%' |
| ENDING (case-sensitive) | firstname like '%' + ?0 |
| ENDING (case-insensitive) | LOWER(firstname) like '%' + LOWER(?0) |
| CONTAINING (case-sensitive) | firstname like '%' + ?0 + '%' |
| CONTAINING (case-insensitive) | LOWER(firstname) like '%' + LOWER(?0) + '%' |

**PAGINATION**





**Customization**



**@Transactional**

By default, CRUD methods on repository instances are transactional. For read operations, the transaction configuration readOnly flag is set to true. All others are configured with a plain @Transactional so that default transaction configuration applies. For details, see JavaDoc of [SimpleJpaRepository](https://docs.spring.io/spring-data/data-jpa/docs/current/api/index.html?org/springframework/data/jpa/repository/support/SimpleJpaRepository.html). If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:

*Example 98. Custom transaction configuration for CRUD*

public interface UserRepository extends CrudRepository<User, Long> {

@Override

@Transactional(timeout = 10)

public List<User> findAll();

// Further query method declarations

}

Doing so causes the findAll() method to run with a timeout of 10 seconds and without the readOnly flag.

Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations. The following example shows how to use such a facade for more than one repository:

*Example 99. Using a facade to define transactions for multiple repository calls*

@Service

class UserManagementImpl implements UserManagement {

private final UserRepository userRepository;

private final RoleRepository roleRepository;

@Autowired

public UserManagementImpl(UserRepository userRepository,

RoleRepository roleRepository) {

this.userRepository = userRepository;

this.roleRepository = roleRepository;

}

@Transactional

public void addRoleToAllUsers(String roleName) {

Role role = roleRepository.findByName(roleName);

for (User user : userRepository.findAll()) {

user.addRole(role);

userRepository.save(user);

}

}

This example causes call to addRoleToAllUsers(…) to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used. Note that you must activate <tx:annotation-driven /> or use @EnableTransactionManagement explicitly to get annotation-based configuration of facades to work. This example assumes you use component scanning.

5.7.1. Transactional query methods

To let your query methods be transactional, use @Transactional at the repository interface you define, as shown in the following example:

*Example 100. Using @Transactional at query methods*

@Transactional(readOnly = true)

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByLastname(String lastname);

@Modifying

@Transactional

@Query("delete from User u where u.active = false")

void deleteInactiveUsers();

}

Typically, you want the readOnly flag to be set to true, as most of the query methods only read data. In contrast to that, deleteInactiveUsers() makes use of the @Modifying annotation and overrides the transaction configuration. Thus, the method runs with the readOnly flag set to false.

For more:

<https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/transaction.html>

https://dzone.com/articles/how-does-spring-transactional

**Locking**

To specify the lock mode to be used, you can use the @Lock annotation on query methods, as shown in the following example:

*Example 101. Defining lock metadata on query methods*

interface UserRepository extends Repository<User, Long> {

// Plain query method

@Lock(LockModeType.READ)

List<User> findByLastname(String lastname);

}

This method declaration causes the query being triggered to be equipped with a LockModeType of READ. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the @Lock annotation, as shown in the following example:

*Example 102. Defining lock metadata on CRUD methods*

interface UserRepository extends Repository<User, Long> {

// Redeclaration of a CRUD method

@Lock(LockModeType.READ);

List<User> findAll();

}

**Transaction Isolation levels**

## **Introduction**

Transaction isolation level is a concept that is not exclusive to the Spring framework. It is applied to transactions in general and is directly related with the ACID transaction properties. Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

## **Usage example**

Using the **@Transactional** annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level:

**Isolation level in a transactional method**

@Autowired

private TestDAO testDAO;

@Transactional(isolation=Isolation.READ\_COMMITTED)

public void someTransactionalMethod(User user) {

// Interact with testDAO

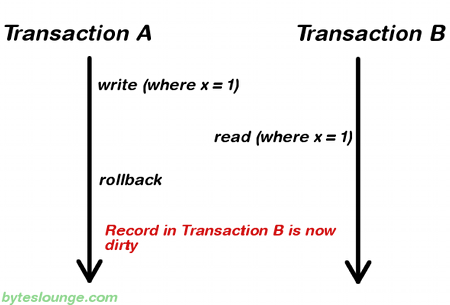
}

We are defining this method to be executed in a transaction which isolation level is **READ\_COMMITTED**. We will see each isolation level in detail in the next sections.

## **READ\_UNCOMMITTED**

**READ\_UNCOMMITTED** isolation level states that a transaction **may** read data that is still **uncommitted** by other transactions. This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like **dirty reads**. Let's see the following image:

**Dirty read**



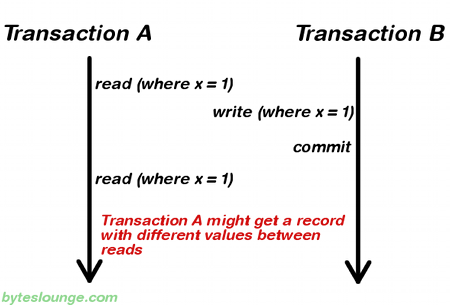
In this example **Transaction A** writes a record. Meanwhile **Transaction B** reads that same record before **Transaction A** commits. Later **Transaction A** decides to rollback and now we have changes in **Transaction B** that are inconsistent. This is a **dirty read**. **Transaction B** was running in **READ\_UNCOMMITTED** isolation level so it was able to read **Transaction A** changes before a commit occurred.

**Note:**READ\_UNCOMMITTED is also vulnerable to **non-repeatable reads** and **phantom reads**. We will also see these cases in detail in the next sections.

## **READ\_COMMITTED**

**READ\_COMMITTED** isolation level states that a transaction can't read data that is **not** yet committed by other transactions. This means that the **dirty read** is no longer an issue, but even this way other issues may occur. Let's see the following image:

**Non-repeatable read**



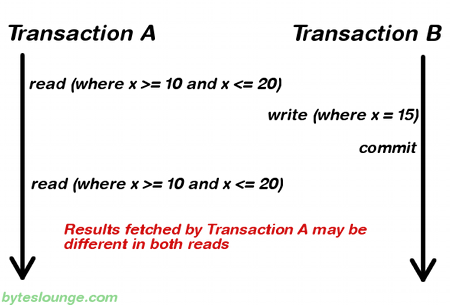
In this example **Transaction A** reads some record. Then **Transaction B** writes that same record and commits. Later **Transaction A** reads that same record again and may get different values because **Transaction B** made changes to that record and committed. This is a **non-repeatable read**.

**Note:**READ\_COMMITTED is also vulnerable to **phantom reads**. We will also see this case in detail in the next section.

## **REPEATABLE\_READ**

**REPEATABLE\_READ** isolation level states that if a transaction reads one record from the database multiple times the result of all those reading operations must always be the same. This eliminates both the **dirty read** and the **non-repeatable read** issues, but even this way other issues may occur. Let's see the following image:

**Phantom read**



In this example **Transaction A** reads a **range** of records. Meanwhile **Transaction B** inserts a new record in the same range that **Transaction A** initially fetched and commits. Later **Transaction A** reads the same range again and will also get the record that **Transaction B** just inserted. This is a **phantom read**: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

## **SERIALIZABLE**

**SERIALIZABLE** isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (**read**, **range** and **write** locking) so they appear as if they were executed in a serialized way. This leads to a scenario where **none** of the issues mentioned above may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

## **DEFAULT**

**DEFAULT** isolation level, as the name states, uses the default isolation level of the datastore we are actually connecting from our application.

## **Summary**

To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **dirty reads** | **non-repeatable reads** | **phantom reads** |
| **READ\_UNCOMMITTED** | yes | yes | yes |
| **READ\_COMMITTED** | no | yes | yes |
| **REPEATABLE\_READ** | no | no | yes |
| **SERIALIZABLE** | no | no | no |