[General Questions 1](#_Toc68382610)

[1) What is Spring data JPA 2](#_Toc68382611)

[2) Repository in spring data jpa? 2](#_Toc68382612)

[3) Configuration steps of spring data jpa without spring boot? 3](#_Toc68382613)

[4) How to create custom repo? 4](#_Toc68382614)

[5) CrudRepository vs Repository? 4](#_Toc68382615)

[6) @NoRepositoryBean 5](#_Toc68382616)

[7) What are queryMethods? 5](#_Toc68382617)

[8) Rules for queryMethod? 5](#_Toc68382618)

[9) When should we use query generation methods of Spring? 7](#_Toc68382619)

[10) Create Query Methods using @Query 7](#_Toc68382620)

[11) What does query method return? 8](#_Toc68382621)

[12) How to pass parameter to query parameter? 9](#_Toc68382622)

[13) How to create JPQL query using @Query? 11](#_Toc68382623)

[14) How to create SQL query using @Query? 11](#_Toc68382624)

[15) When should we use @Query method? 13](#_Toc68382625)

[16) How to use @NamedQuery or @NativeNamedQuery? 14](#_Toc68382626)

[17) When should we use Named Query? 16](#_Toc68382627)

[18) SQLResultMapping 17](#_Toc68382628)

[19) Criteria API 17](#_Toc68382629)

[20) Sorting 17](#_Toc68382630)

[21) Pagination 19](#_Toc68382631)

[22) Auditing in spring data JPA 20](#_Toc68382632)

[23) Important methods of CRUDRepo 20](#_Toc68382633)

[24) Spring JPA Advantage 20](#_Toc68382634)

[25) How to enable spring data JPa? 21](#_Toc68382635)

[26) How to define Query Methods for the nested property. 21](#_Toc68382636)

[27) Difference between CrudRepository and JpaRepository in Spring Data JPA? 23](#_Toc68382637)

[28) Repository vs CurdRepository 23](#_Toc68382638)

[29) How CrudRepository save() methods internally works in Spring Data JPA? 24](#_Toc68382639)

[30) Tell something about the CrudRepository saveAll() method. 25](#_Toc68382640)

[31) How to implement projection using Spring Data JPA? 25](#_Toc68382641)

[32) How to write query using @NamedQueries and @NamedNativeQueries in Spring Data JPA. 27](#_Toc68382642)

[33) Difference between findById() and getOne() in Spring Data JPA? 30](#_Toc68382643)

[34) Difference between delete() vs deleteInBatch() Methods in Spring Data JPA. 30](#_Toc68382644)

[35) Difference between deleteAll() Vs deleteAllInBatch() in Spring Data JPA. 31](#_Toc68382645)

[36) How to use Named Query? 34](#_Toc68382646)

[37) What will happen if we define wrong query methods? 34](#_Toc68382647)

[38) How to deine case insensitive search? 35](#_Toc68382648)

[39) Output of below query? 35](#_Toc68382649)

[40) List important keyword for QueryMethod 36](#_Toc68382650)

[REFERENCES: 36](#_Toc68382651)

# General Questions

## What is Spring data JPA

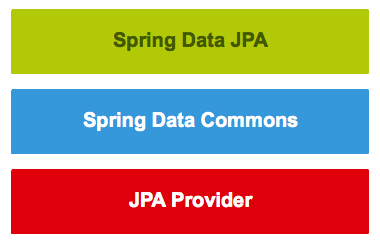
### Spring Data JPA is one of Spring Data module which provides **predefined repository methods to perform CRUD operation**. Using Spring Data JPA we define the repository interface and query methods (query creation from method names) to access the data from the database. It makes easier to build Spring applications that use data access technologies.

### 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:

### [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) provides support for creating JPA repositories by extending the Spring Data repository interfaces.

### [Spring Data Commons](https://github.com/spring-projects/spring-data-commons) provides the infrastructure that is shared by the datastore specific [Spring Data projects](http://projects.spring.io/spring-data/).

### The JPA Provider implements the Java Persistence API.



## Repository in spring data jpa?

### We can use Spring Data JPA without paying any attention to the actual implementation of the repository abstraction, but we have to be familiar with the Spring Data repository interfaces. These interfaces are described in the following:

### **First**, the **Spring Data Commons** project provides the following interfaces:

### The [**Repository<T, ID extends Serializable>**](http://docs.spring.io/spring-data/commons/docs/current/api/index.html?org/springframework/data/repository/Repository.html) interface is a marker interface that has two purposes:

### 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 during classpath scanning.

### The [CrudRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html" \t "_blank) interface provides CRUD operations for the managed entity.

### The [PagingAndSortingRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html" \t "_blank) interface declares the methods that are used to sort and paginate entities that are retrieved from the database.

### The [QueryDslPredicateExecutor<T>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/querydsl/QueryDslPredicateExecutor.html" \t "_blank) interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [QueryDsl](http://www.querydsl.com/" \t "_blank) Predicate objects.

### **Second**, the **Spring Data JPA** project provides the following interfaces:

### The [JpaRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html" \t "_blank) interface is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.

### The [JpaSpecificationExecutor<T>](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaSpecificationExecutor.html" \t "_blank) interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [Specification<T>](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/Specification.html) objects that use the JPA criteria API.

### The repository hierarchy looks as follows:

### [springdatajrepositories](https://www.petrikainulainen.net/wp-content/uploads/springdatajrepositories.png)

## Configuration steps of spring data jpa without spring boot?

### We can create the configuration class, which configures the persistence layer of a Spring application, by following these steps:

### Create the properties file that contains the properties used by our application context configuration class.

### Configure the datasource bean.

### Configure the entity manager factory bean.

### Configure the transaction manager bean.

### Enable annotation-driven transaction management.

### Configure Spring Data JPA.

## How to create custom repo?

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

public List<Student> findByNameAndRollNumber(String name, String rollNumber);

public List<Student> findByNameOrRollNumber(String name, String rollNumber);

public List<Student> findByNameAndRollNumberOrUniversity(String name, String rollNumber, String university);

}

### We can create the repository that provides CRUD operations for *Todo* objects by using one of the following methods:

### Create an interface that extends the *[CrudRepository](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html" \t "_blank)* interface.

### Create an interface that extends the [*Repository*](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/Repository.html) interface and add the required methods to the created interface.

## CrudRepository vs Repository?

Crud provides basic method to find, delete and save entity. Repository does not provide any method it is just marker.

### If we want to expose all repository methods that are declared by the *CrudRepository* interface **AND** we don’t want to return *Optional* (Guava / Java 8) objects, our repository interfaces should extend the *CrudRepository* interface.

### If we don’t want to expose all repository methods that are declared by the *CrudRepository* interface **OR** we want to return *Optional* (Guava / Java 8) objects, our repository interfaces must extend the *Repository* interface.

### When we create an interface, we should not add unnecessary methods to it. We should keep the interface as small as possible because small interfaces are easier to use and they help us to create components that have only one job.

### *Optional* helps us to create better APIs because it reveals that there might not be a return value.

## @NoRepositoryBean

### Annotate the created interface with the @NoRepositoryBean annotation. This ensures that Spring Data JPA doesn’t try to create an implementation for our base repository interface.

### @NoRepositoryBean

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

## What are queryMethods?

### Query methods are methods that find information from the database and are declared on the repository interface. For example, if we want to create a database query that finds the *Todo* object that has a specific id, we can create the query method by adding the *findById()* method to the *TodoRepository* interface. After we have done this, our repository interface looks as follows:

|  |  |
| --- | --- |
| 1234567 | import org.springframework.data.repository.Repository;interface TodoRepository extends Repository<Todo, Long> {    //This is a query method.    Todo findById(Long id); |

## Rules for queryMethod?

### Rule 1 – The name of the query method must start with findBy or getBy  or queryBy or countBy or readBy prefix. The findBy is mostly used by the developer.

### For example findByName(String name), getByName(String name), queryByName(String name), countByName(String name), readByName(String name),

### Note – All the above query methods will return all students whose name is “what ever name we provide as parameter”.

### Rule 2 – The first character of field name should capital letter. Although if we write the first character of the field in small then it will work but we should use camelcase for the method name.

### Both are valid query method defined below but we should follow the first way.

### public List<Student> findByName(String name);

### public List<Student> findByname(String name);

### Rule 3 – While using findBy or getBy or queryBy or countBy or readBy the character B must be in capital letter, else we will get an exception while deployment.

### Caused by: org.springframework.data.mapping.PropertyReferenceException: No property querybyName found for type Student!

### Invalid query method.

### public List<Student> findbyName(String name);

### Rule 4 – We can write the query method using multiple fields using predefined keywords(eg. And, Or etc) but these keywords are case sensitive. We must use “And” instead of “and”.

### The name of our query method must start with one of the following prefixes: find…By, read…By, query…By, count…By, and get…By.

### If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word. If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords. For example, findTopBy, findTop1By, findFirstBy, and findFirst1By all return the first entity that matches with the specified search criteria.

### If we want to select unique results, we have to add the Distinct keyword before the first By word. For example, findTitleDistinctBy or findDistinctTitleBy means that we want to select all unique titles that are found from the database.

### We must add the search criteria of our query method after the first By word. We can specify the search criteria by combining [property expressions](http://docs.spring.io/spring-data/jpa/docs/1.8.x/reference/html/#repositories.query-methods.query-property-expressions) with the [supported keywords](http://docs.spring.io/spring-data/jpa/docs/1.8.x/reference/html/#repository-query-keywords).

### If our query method specifies x search conditions, we must add x method parameters to it. In other words, the number of method parameters must be equal than the number of search conditions. Also, the method parameters must be given in the same order than the search conditions.

### We must set the return type of the query method by following the rules that were described in below section.

### public Todo findById(Long id);

### public List<Todo> findByTitleOrDescription(String title, String description);

### public List<Todo> findDistinctByTitle(String title);

### public long countByTitle(String title);

### public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);

List<Todo> findByDescriptionContainsOrTitleContainsAllIgnoreCase(String descriptionPart, String titlePart);

### 

## When should we use query generation methods of Spring?

### This query generation strategy has the following benefits:

### Creating simple queries is fast.

### The method name of our query method describes the selected value(s) and the used search condition(s).

### This query generation strategy has the following weaknesses:

### The features of the method name parser determine what kind of queries we can create. If the method name parser doesn’t support the required keyword, we cannot use this strategy.

### The method names of complex query methods are long and ugly.

### There is no support for dynamic queries.

### When we think about the pros and cons of this strategy and take a second look at our examples, it becomes clear that the length of our query method name determines whether or not we should use this strategy.

### I am not a big fan of “super long” method names because they become unreadable very fast. If we compare the method name: *findById* with the method name: *findByDescriptionContainsOrTitleContainsAllIgnoreCase*, we notice that first one is very easy to read. The second method name is not nearly as easy to read as the first one, but it is not impossible to read either (yet). It is a borderline case.

### Because I want to write code that is easy to read, I think that we should use this strategy only when we are creating simple queries that have only one or two search conditions.

## Create Query Methods using @Query

### We can configure the invoked database query by annotating the query method with the *@Query* annotation. It supports both JPQL and SQL queries, and the query that is specified by using the *@Query* annotation precedes all other query generation strategies.

### In other words, if we create a query method called *findbyId()* and annotate it with the *@Query* annotation, Spring Data JPA won’t (necessarily) find the entity whose *id* property is equal than the given method parameter. It invokes the query that is configured by using the *@Query* annotation.

|  |
| --- |
| @Query("SELECT t FROM Todo t WHERE t.title = 'title'")      public List<Todo> findById();  } |

### Even though the *findById()* method follows the naming convention that is used to create database queries from the method name of the query method, the *findById()* method returns todo entries whose title is ‘title’, because that is the query which is specified by using the *@Query* annotation.

## What does query method return?

### A query method can return only one result or more than one result. Also, we can create a query method that is invoked asynchronously

FOR single result:

### Basic type. Our query method will return the found basic type or null.

### Entity. Our query method will return an entity object or null.

### Guava / Java 8 Optional<T>. Our query method will return an Optional that contains the found object or an empty Optional.

### Here are some examples of query methods that return only one result:

|  |  |
| --- | --- |
| 1234567891011121314151617 | import java.util.Optional;import org.springframework.data.jpa.repository.Query;import org.springframework.data.repository.Repository;import org.springframework.data.repository.query.Param;interface TodoRepository extends Repository<Todo, Long> {    @Query("SELECT t.title FROM Todo t where t.id = :id")    String findTitleById(@Param("id") Long id);    @Query("SELECT t.title FROM Todo t where t.id = :id")    Optional<String> findTitleById(@Param("id") Long id);    Todo findById(Long id);    Optional<Todo> findById(Long id) |

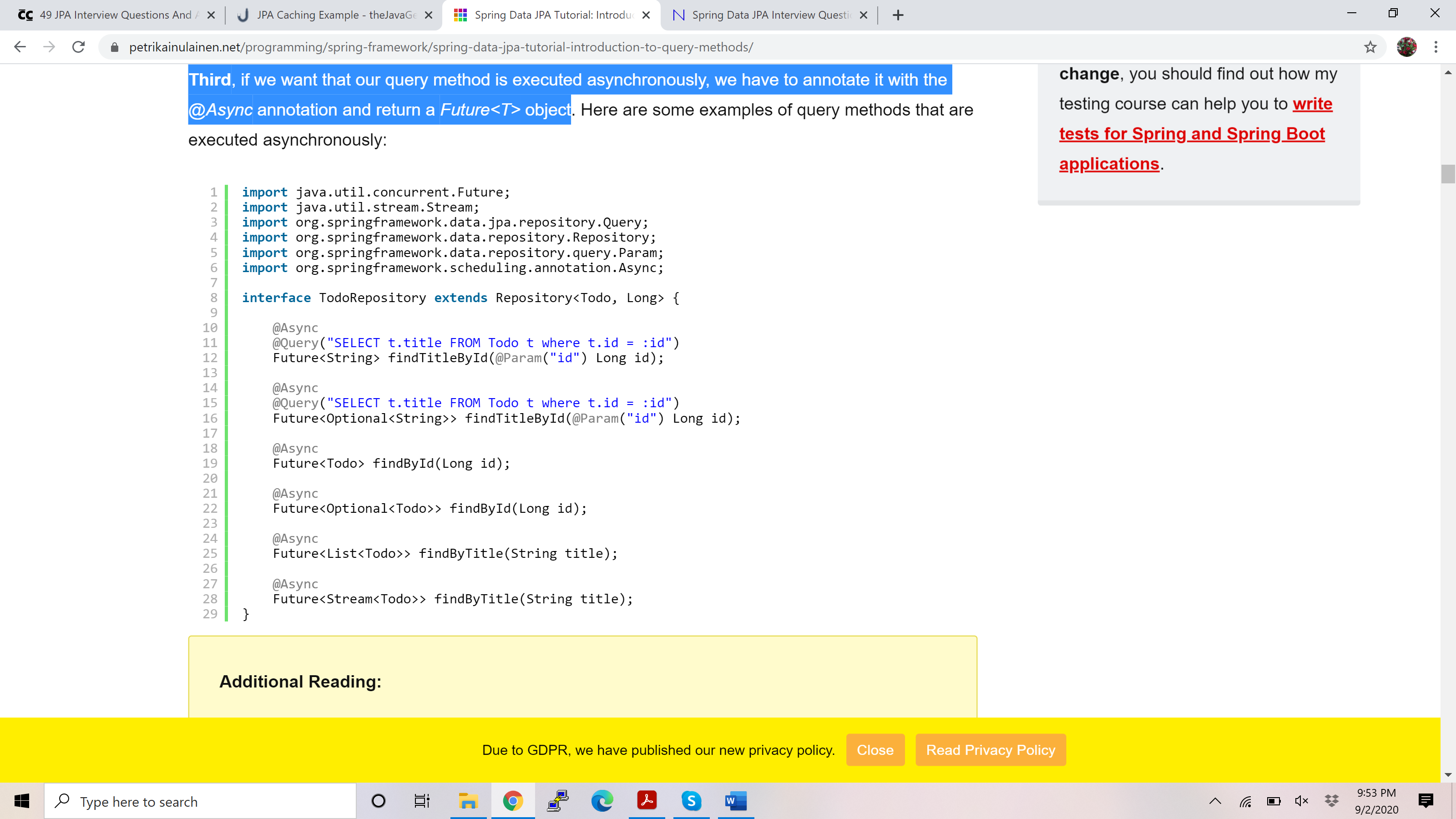
### **Second**, if we are writing a query method that should return more than one result, we can return the following types:

### *List<T>*. Our query method will return a list that contains the query results or an empty list.

### *Stream<T>*. Our query method will return a *Stream* that can be used to access the query results or an empty *Stream*.

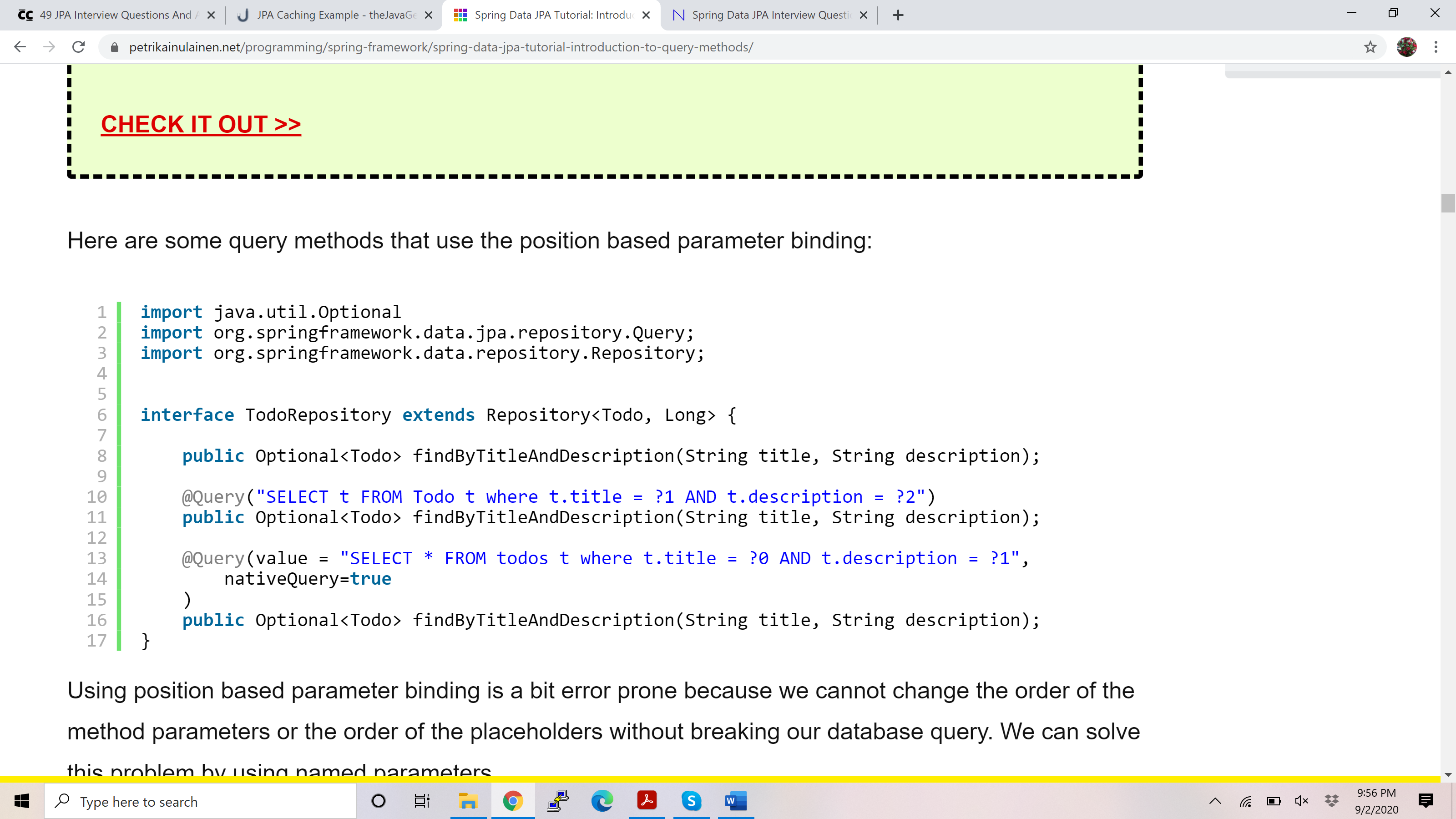
* import java.util.stream.Stream;
* import org.springframework.data.repository.Repository;
* interface TodoRepository extends Repository<Todo, Long> {
* List<Todo> findByTitle(String title);
* Stream<Todo> findByTitle(String title);
* }

### **Third**, if we want that our query method is executed asynchronously, we have to annotate it with the @Async annotation and return a Future<T> object



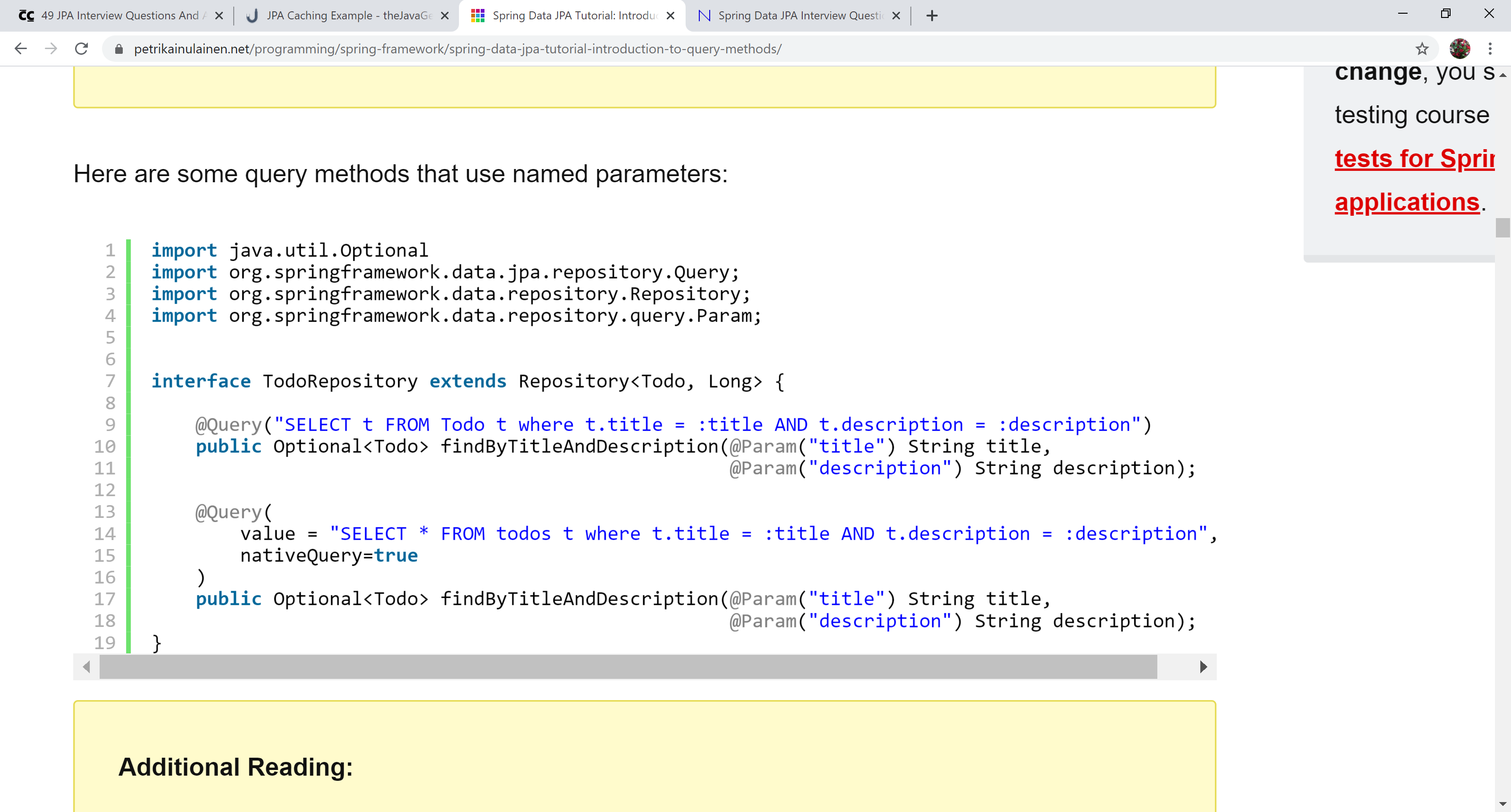
## How to pass parameter to query parameter?

### Spring Data JPA supports both position based parameter binding and named parameters. Both of these options are described in the following.

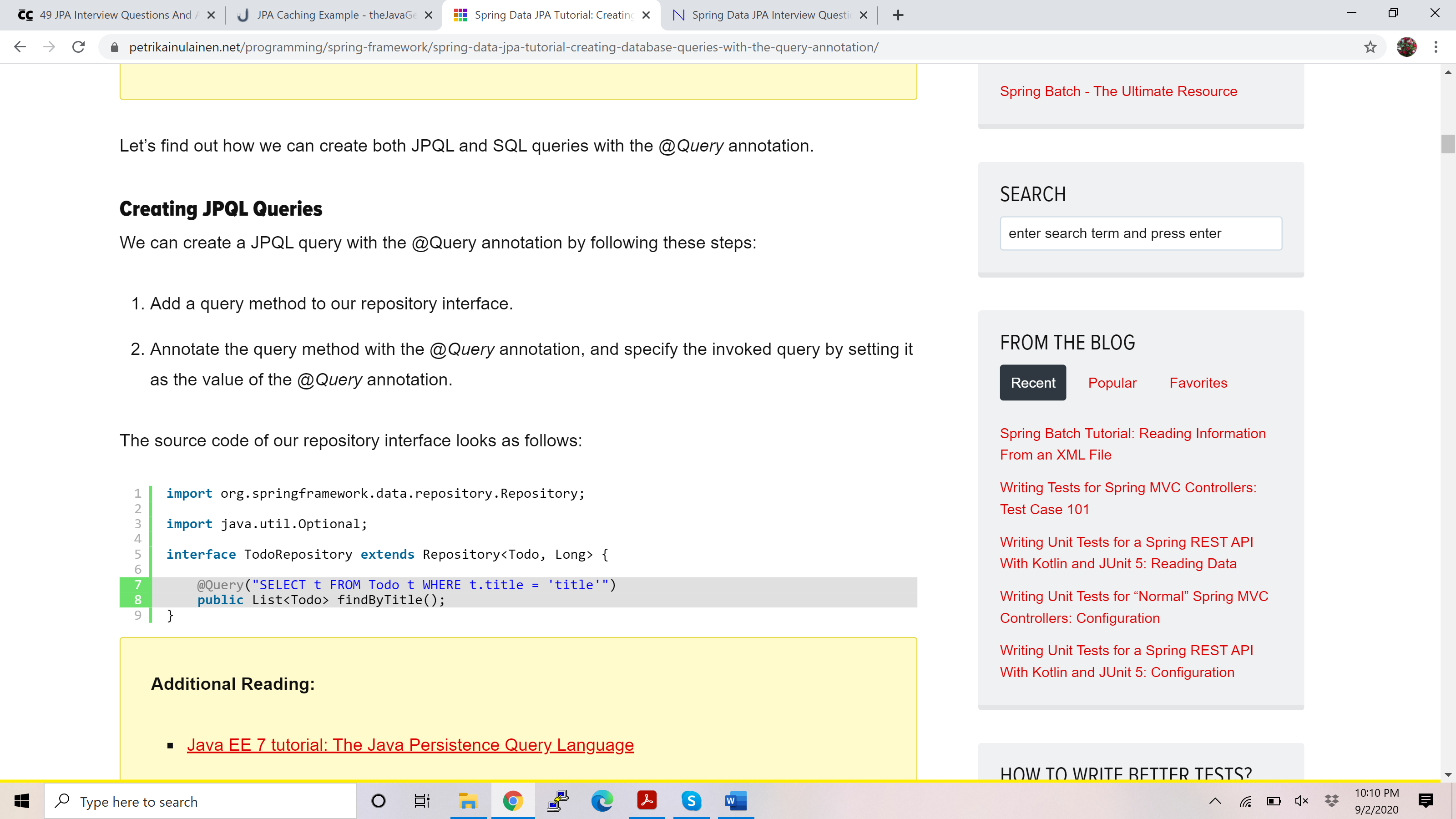


### Using position based parameter binding is a bit error prone because we cannot change the order of the method parameters or the order of the placeholders without breaking our database query. We can solve this problem by using named parameters.

### We can use named parameters by replacing the numeric placeholders found from our database queries with concrete parameter names, and annotating our method parameters with the @Param annotation.



## How to create JPQL query using @Query?



## How to create SQL query using @Query?

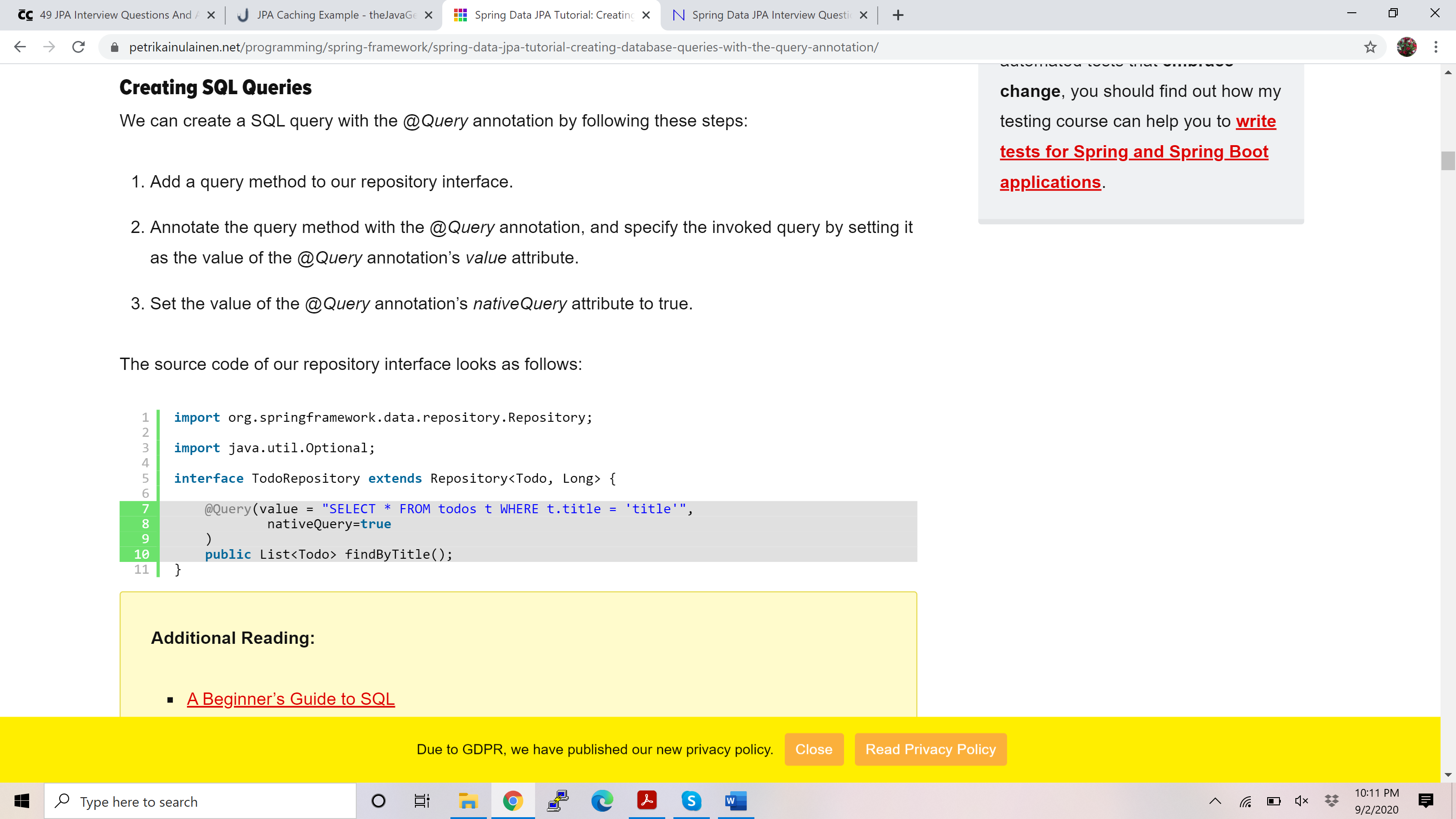
### We can create a SQL query with the *@Query* annotation by following these steps:

### Add a query method to our repository interface.

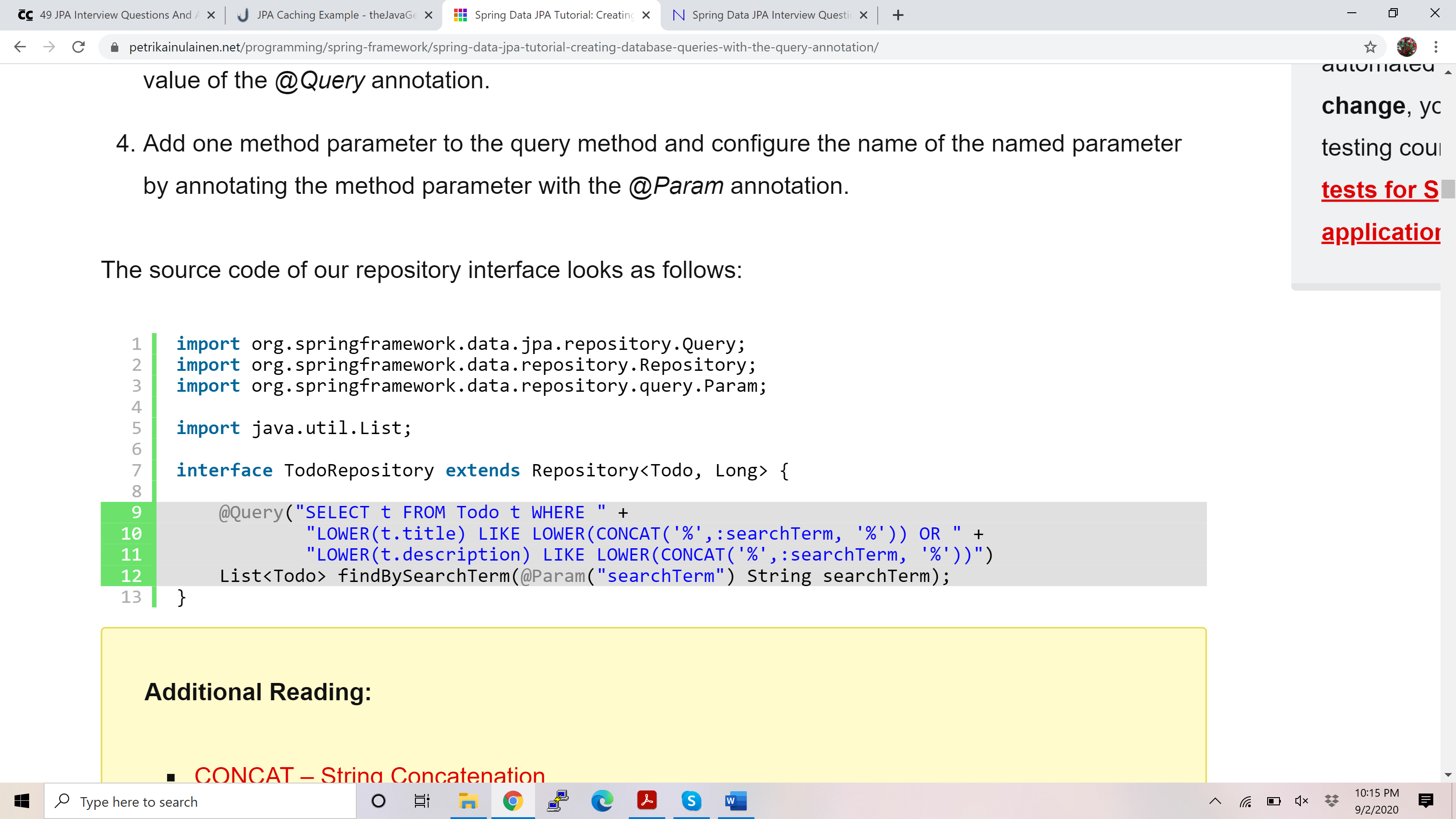
### Annotate the query method with the *@Query* annotation, and specify the invoked query by setting it as the value of the *@Query* annotation’s *value* attribute.

### Set the value of the *@Query* annotation’s *nativeQuery* attribute to true.

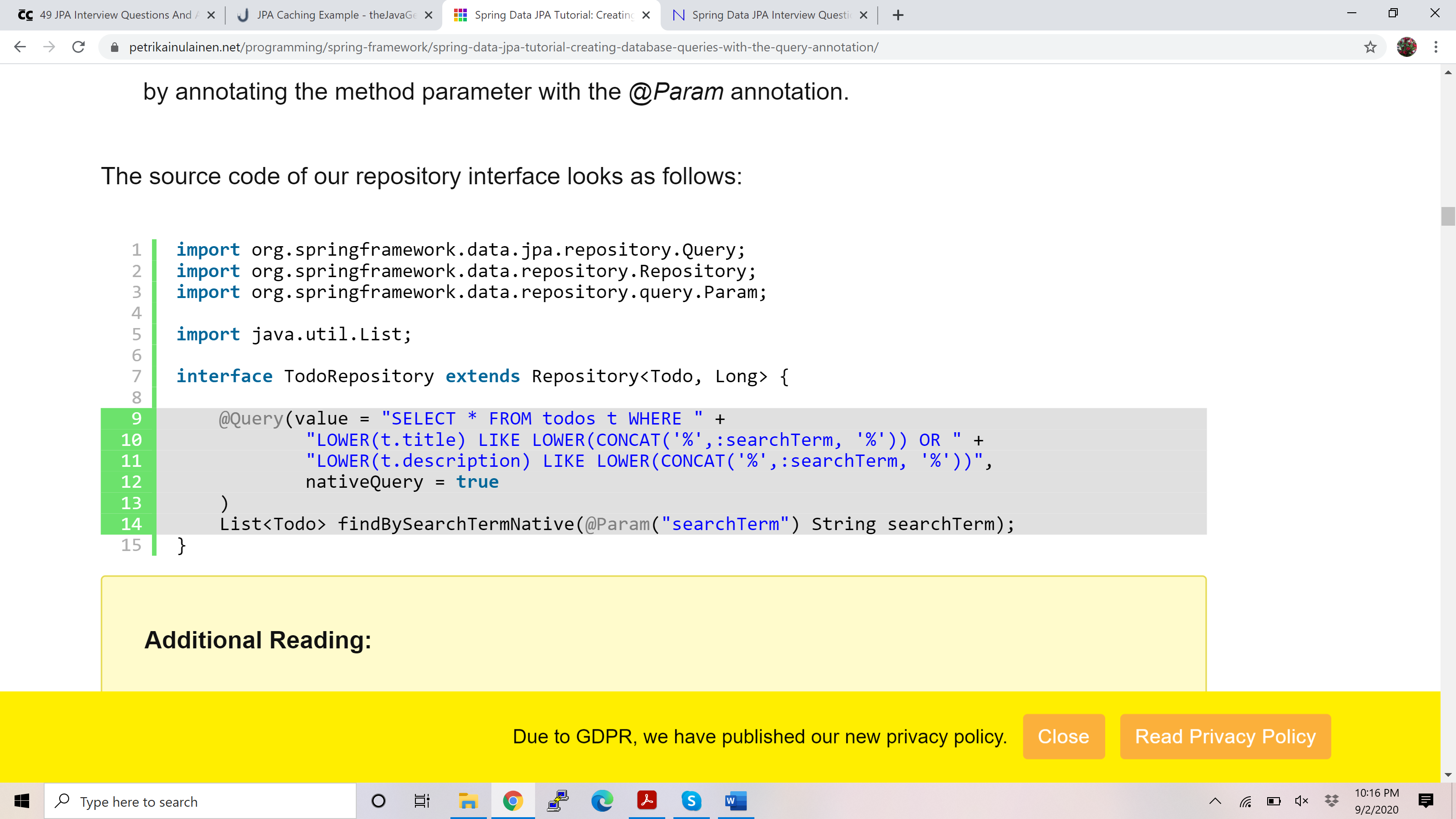
### The source code of our repository interface looks as follows:



Example JPQL:



Example SQL:



## When should we use @Query method?

### The @Query annotation has the following benefits:

### It supports both JPQL and SQL.

### The invoked query is found above the query method. In other words, it is easy to find out what the query method does.

### There is no naming convention for query method names.

### The @Query annotation has the following drawbacks:

### There is no support for dynamic queries.

### If we use SQL queries, we cannot change the used database without testing that our SQL queries work as expected.

### When we think about the pros and cons of the @Query annotation, it becomes clear that the query methods that use this strategy aren’t as easy to write or read as the query methods that use the [query generation from the method name strategy](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-from-method-names/).

### However, these query methods have two crucial advantages:

### If we need to find out what database query is invoked by our query method, we can find the invoked query above the query method.

### The code that uses our query methods is easy to read because we don’t have to use long method names.

### Thus, if we don’t want to use the query generation from the method name strategy, because the names of our query methods would be too long, we should use the @Query annotation.

## How to use @NamedQuery or @NativeNamedQuery?

### We can specify named queries with Spring Data JPA by using a properties file, annotations, or the orm.xml file.

### Before can take a closer look at these methods, we have to learn a few rules that we must follow when we are specifying the names of our named queries. These rules are:

### If we want to use the default naming strategy of Spring Data JPA, we have to specify the name of the named query by using this syntax: **[entity class name].[name of the invoked query method].**

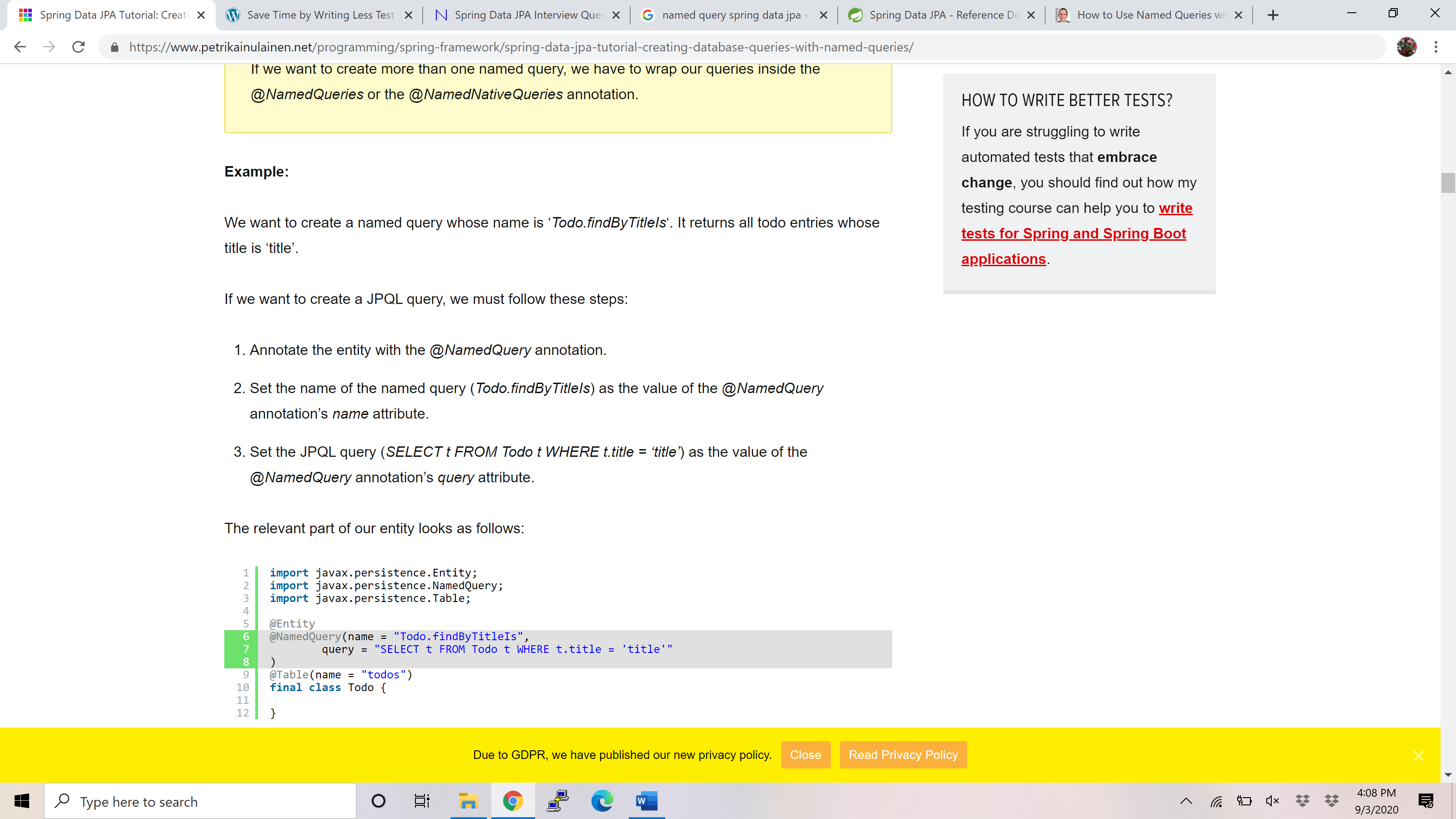
### If we want to use some other syntax, we have to configure the name of named query when we create the query method that invokes it. We can do this by using [the name attribute of the @Query annotation](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/Query.html#name--).

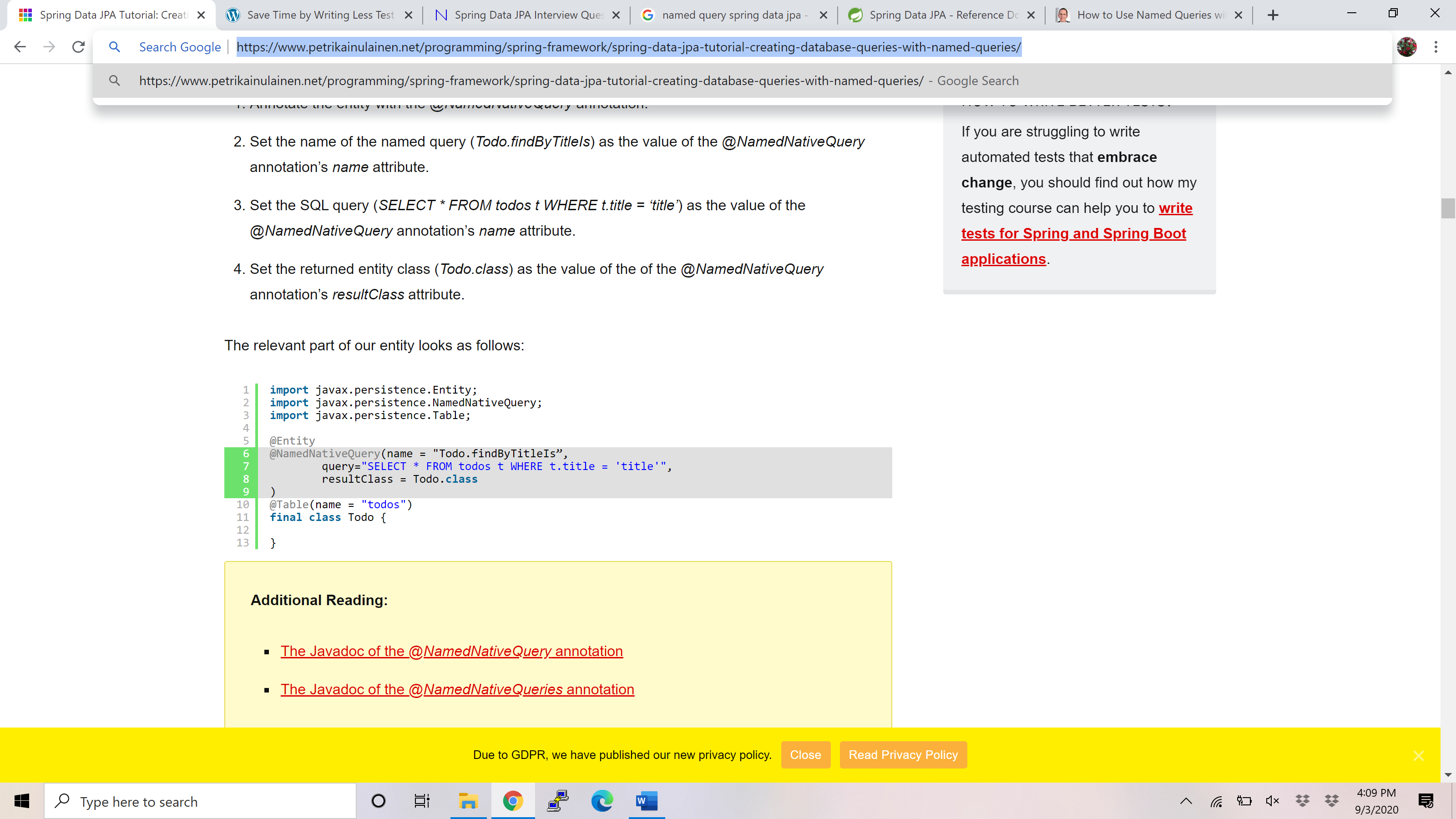
### Propeprties file: We can declare named queries by adding them into the jpa-named-queries.properties file that is found from the META-INF folder of our classpath.

### Annotation: annotate Entity class contains two param, name and value.

### If we want to create a JPQL query, we have to annotate our entity with the @NamedQuery annotation.

### If we want to create a SQL query, we have to annotate our entity with the @NamedNativeQuery annotation.



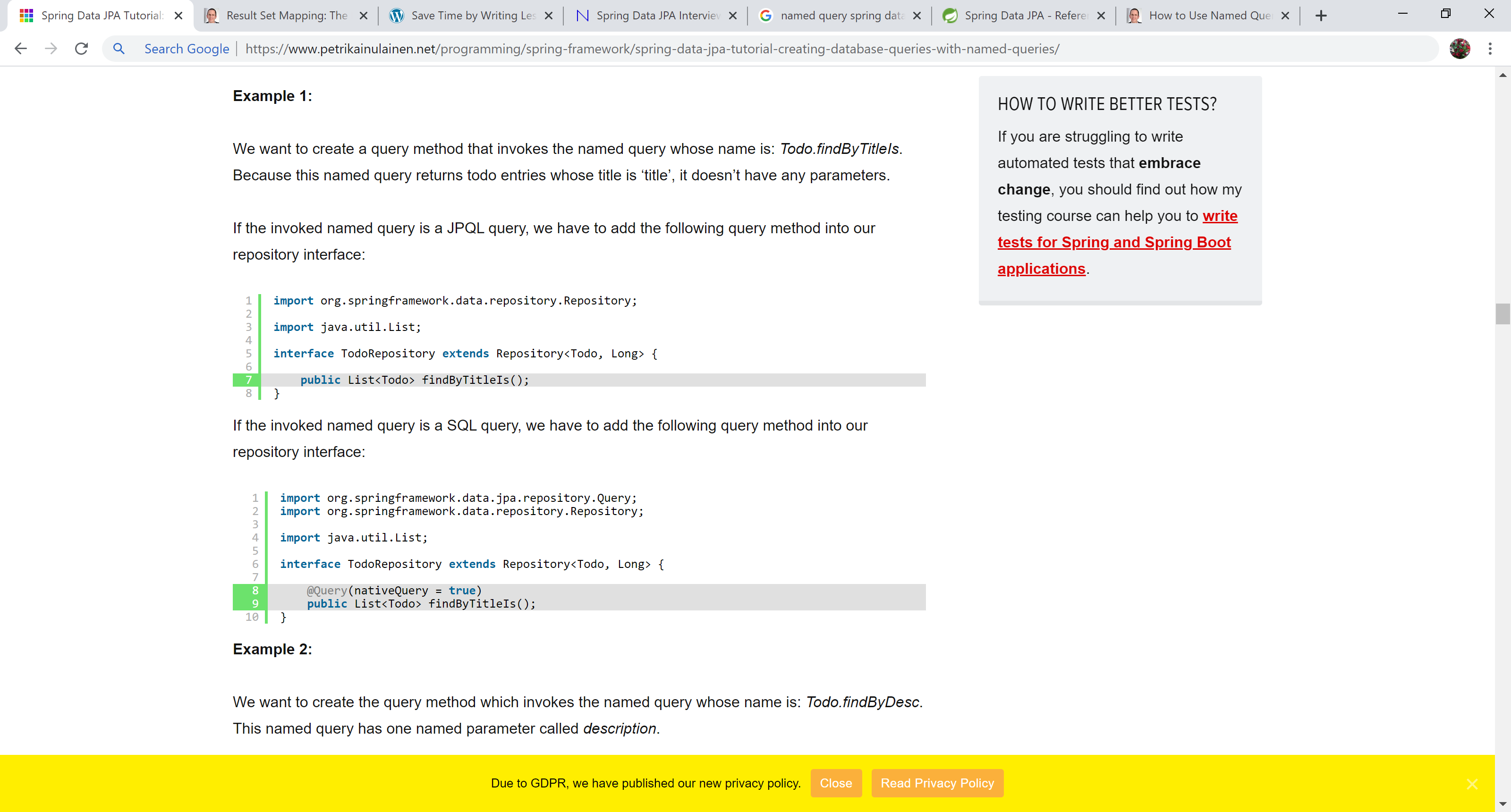


Orm.xml - <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-named-queries/>

### Calling the named query in repository:

We can create the query method that invokes a specific named query by following these steps:

1. Add a query method into our repository interface and follow these rules:
   * If our named query uses the default naming strategy of Spring Data JPA, we must ensure that the name of the query method identifies the invoked named query.
   * If our named query doesn’t use the default naming strategy, we have to annotate the query method with the *@Query* annotation and configure the name of invoked named query by using the *name* attribute of the *@Query* annotation.
2. If the invoked named query is a SQL query, we have to annotate the query method with the *@Query* annotation and set the value of its *nativeQuery* attribute to *true*.
3. Add the correct method parameters to the query method.
4. Specify the return type of the query method.



## When should we use Named Query?

Named queries have the following benefits:

* Named queries support both JPQL and SQL.
* If we have an existing application that uses named queries, it is easy to refactor it to use use Spring Data JPA (if we want to).
* Named queries provide support for SQL result set mapping. This means that we can write complex SQL queries and map the query results into objects.

Named queries have the following drawbacks:

* We cannot see the invoked database query from the repository interface.
* There is no support for dynamic queries.
* If we specify named queries by using annotations, they “litter” the source code of our entity classes.

If we think about the pros and cons of named queries, it becomes clear that the query methods which use named queries aren’t as easy to read or write as the query methods that use either [the query generation from the method name strategy](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-from-method-names/) or [the @Query annotation](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-the-query-annotation/).

However, named queries have two advantages:

1. We can write complex SQL queries and map the query results into objects.
2. If we need to refactor an existing application to use Spring Data JPA, we don’t have to move its named queries to our repository interfaces.

Thus, I think that we should create our query methods by using the query generation from the method name strategy or the @Query annotation.

However, if we cannot create the query method by using these two methods, we can use named queries.

Also, if we are refactoring an existing application to use Spring Data JPA, using its existing named queries is a no-brainer because it can save us a lot of work.

## SQLResultMapping

<https://thorben-janssen.com/result-set-mapping-basics/>

<https://vladmihalcea.com/jpa-sqlresultsetmapping/>

It is used to map the query result to Object for complelx situations where there is not one to one mapping between entity and result set.

## Criteria API

## Sorting

### Sorting can be done in 3 ways –

### Extend repository by PagingAndSortingRepository

### Sorting with query method names:

### If we create our database queries by using the [query generation from the method name strategy](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-from-method-names/), we can sort the query results of our database queries by using the OrderBy keyword. We can use the OrderBy keyword by following these steps:

### Append the OrderBy keyword to the method name of our query method.

### Append the name of the property to the method name of our query method and transform its first letter into uppercase. If we want to order our query results by using the title of a todo entry, we have to append the string: Title to the method name of our query method.

### Describe the sort direction. If we want to sort the query results in ascending order, we have to append the keyword Asc to the method name of our query method. On the other hand, if we want to sort the query results in descending order, we have to append the keyword Desc to the method name of our query method.

### If we need to sort our query results by using multiple properties, we have to go back to the step 2.

### List<Todo> findByTitleOrderByTitleAsc(String title);

### List<Todo> findByTitleOrderByTitleAscDescriptionDesc(String title);

### **Sorting with @Query or @Named or @Native Query**

### Use sorting logic in Query

### SELECT t FROM Todo t WHERE

### LOWER(t.title) LIKE LOWER(CONCAT('%',:searchTerm, '%')) OR

### LOWER(t.description) LIKE LOWER(CONCAT('%',:searchTerm, '%'))

### ORDER BY t.title ASC

### **Using Sort Object**

### Create a sort Object and pass that object to the method parameter of repository.

### Obtain the Sort object that describes the sorting options of the invoked database query.

### Pass the Sort object forward to the correct repository method as a method parameter.

### private Sort orderBy() {

### return new Sort(Sort.Direction.DESC, "title", "description");

### }

### Pass this Object to FindAll (Sort sort) or other methods.

## Pagination

### This can be done by extending repo to PagingAndSortingRepositoy.

Once we have our repository extending from *PagingAndSortingRepository*, we just need to:

* Create or obtain a *PageRequest* object, which is an implementation of the *Pageable* interface
* Pass the *PageRequest* object as an argument to the repository method we intend to use
* We can create a *PageRequest* object by passing in the requested page number and the page size.

Here, the page counts starts at zero:

|  |  |
| --- | --- |
|  | Pageable firstPageWithTwoElements = PageRequest.of(0, 2); |
|  |  |
|  | Pageable secondPageWithFiveElements = PageRequest.of(1, 5); |

Once we have our *PageRequest* object\, we can pass it in while invoking our repository's method:

|  |  |
| --- | --- |
|  | Page<Product> allProducts = productRepository.findAll(firstPageWithTwoElements); |
|  |  |
|  | List<Product> allTenDollarProducts = |
|  | productRepository.findAllByPrice(10, secondPageWithFiveElements); |

The *findAll(Pageable pageable)* method by default returns a *Page<T>* object.

However, **we can choose to return either a *Page<T>,*a*Slice<T>* or a *List<T>* from any of our custom methods returning a paginated data**.

A *Page<T>* instance, in addition to having the list of *Product*s, also knows about the total number of available pages. **It triggers an additional count query to achieve it. To avoid such an overhead cost, we can instead return a *Slice<T>* or a *List<T>*.**

A *Slice* only knows about whether the next slice is available or not.

Similarly, to just have our query results sorted, we can simply [pass an instance of Sort](https://www.baeldung.com/spring-data-sorting) to the method:

|  |  |
| --- | --- |
|  | Page<Product> allProductsSortedByName = productRepository.findAll(Sort.by(**"name"**)); |

However, what if we want to**both sort and page our data?**

We can do that by passing the sorting details into our PageRequest object itself:

|  |  |
| --- | --- |
|  | Pageable sortedByName = |
|  | PageRequest.of(0, 3, Sort.by(**"name"**)); |
|  |  |
|  | Pageable sortedByPriceDesc = |
|  | PageRequest.of(0, 3, Sort.by(**"price"**).descending()); |
|  |  |
|  | Pageable sortedByPriceDescNameAsc = |
|  | PageRequest.of(0, 5, Sort.by(**"price"**).descending().and(Sort.by(**"name"**))); |

### Based on our sorting requirements, **we can specify the sort fields and the sort direction** while creating our PageRequest instance.

### As usual, we can then pass this Pageable type instance to the repository's method.

<https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-seven-pagination/>

## Auditing in spring data JPA

## Important methods of CRUDRepo

### save(S entity) – Used to save a single entity at a time. See an example of the save() method here.

### saveAll(Iterable<S> – we can save multiple entities at a time. See a complete example of the CrudRepository saveAll() method from scratch here.

### findById(ID id) – Use to get entity basis of id. See an example of the findById() method here.

### existsById(ID id) – Used to check whether an entity is already exited in DB for given Id. See an example of the existsById() method here.

### findAll() – Find all entity of particular type. See an example of the findAll() here.

### findAllById(Iterable<ID> ids) – Return all entity of given ids. See an example here.

### count() – Returns the number of entities. See an example of the count() method here.

### deleteById(ID id) – Delete the entity on basis of id. See a complete example deleteById() method here.

### delete(T entity) – delete the entity which one we are passing. See a complete example here.

### delete(Iterable<? extends T> entities) – delete multiple entities which we are passing. See a complete example here.

### deleteAll() – delete all entities. See a complete example here.

## Spring JPA Advantage

### No need to implement repo in case of simple task

### Paging and sorting available

### JPQL and SQL support

### Support for @Query, @Named @Native

## How to enable spring data JPa?

### First, we need to define some configuration class let’s say JpaConfig.java and then we need to use @EnableJpaRepositories annotation with that class. Also, provide basePackages value where (we have our repositories). Let’s see below example.

### package com.netsurfingzone.config;

### 

### import org.springframework.context.annotation.Configuration;

### import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

### 

### @Configuration

### @EnableJpaRepositories(basePackages = "com.netsurfingzone.repository")

### public class JpaConfig {

### 

### }

## How to define Query Methods for the nested property.

### Consider we have two entities Student.java and Address.java. Student and Address entities are in one to one relationship and we want to fetch all students from the database who belongs to city pune.

### Student.java

### package com.netsurfingzone.entity;

### @Entity

### public class Student {

### @Id

### @GeneratedValue(strategy = GenerationType.AUTO)

### private int id;

### @Column(name = "name")

### private String name;

### @Column(name = "roll\_number")

### private String rollNumber;

### @Column(name = "university")

### String university;

### @OneToOne(cascade = CascadeType.ALL)

### @JoinColumn(name = "address\_id")

### Address address;

### }

### Address.java

### package com.netsurfingzone.entity;

### @Entity

### public class Address {

### @Id

### @GeneratedValue(strategy = GenerationType.AUTO)

### private int id;

### @Column(name = "house\_number")

### private String houseNumber;

### 

### @Column(name = "city")

### private String city;

### 

### }

### Now we want all Students records which belong to city pune.

### Query method to fetch all students who belong to city pune using Spring Data JPA.

### List<Student> findByAddressCity(String city);

### See an example here.

## Difference between CrudRepository and JpaRepository in Spring Data JPA?



## Repository vs CurdRepository

### Repository interface.

### The Repository is a top-level interface in hierarchy.The Repository is a marker interface. It doesn’t have any method.Repository interface has been defined as below.

### @Indexed

### public interface Repository<T, ID> {

### }

### CrudRepository interface.

### The CrudRepository extends Repository interface. It has below methods to perform CRUD operation.

### save(S entity)

### saveAll(Iterable<S> entities)

### findById(ID id)

### existsById(ID id)

### findAll()

### findAllById(Iterable<ID> ids)

### count()

### deleteById(ID id)

### delete(T entity)

### delete(Iterable<? extends T> entities)

### deleteAll()

## How CrudRepository save() methods internally works in Spring Data JPA?

### The CrudRepository’s save() method is used to perform save as well as update operation both. The implementation has been given in SimpleJpaRepository.java, where persist() and merge() is getting called. If we try to save entity first time then persist() method will get invoked and if we try to update the same entity merge() will get invoked.

### public S save(S entity) {

### 

### if (entityInformation.isNew(entity)) {

### em.persist(entity);

### return entity;

### } else {

### return em.merge(entity);

### }

### }

### See an example here.

## Tell something about the CrudRepository saveAll() method.

### The CrudRepository saveAll() method used to save multiple entities and internally annotated with @Transactional annotation. It internally uses save() method only as below.

### @Transactional

### public <S extends T> List<S> saveAll(Iterable<S> entities) {

### List<S> result = new ArrayList<S>();

### for (S entity : entities) {

### result.add(save(entity));

### }

### return result;

### }

### See an example here.

## How to implement projection using Spring Data JPA?

### Consider we have an entity called Student.java as below.

### package com.netsurfingzone.entity;

### @Entity

### public class Student {

### @Id

### @GeneratedValue(strategy = GenerationType.AUTO)

### private int id;

### @Column(name = "name")

### private String name;

### @Column(name = "roll\_number")

### private String rollNumber;

### @Column(name = "university")

### String university;

### 

### }

### and we have some record in the database a.We want to fetch any single column. For example, we want all student name as below. Let’s see how to define Projection for the above requirement.

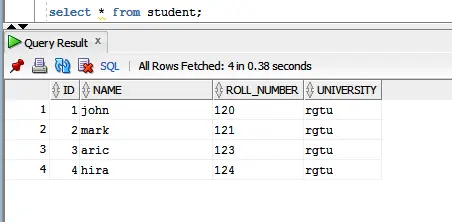
### Define Interface with some method.

### package com.netsurfingzone.repository;

### public interface StudentName {

### String getName();

### }



### Define Repository as below.

### @Repository

### public interface StudentRepository extends JpaRepository<Student, Serializable> {

### 

### 

### List<StudentName> findAllBy();

### }

### Generated query.

### Hibernate:

### select

### student0\_.name as col\_0\_0\_

### from

### student student0\_

## How to write query using @NamedQueries and @NamedNativeQueries in Spring Data JPA.

### Both @NamedQueries and @NamedNativeQueries annotations used with the entity.

### Using @NamedQueries.

### package com.netsurfingzone.entity;

### 

### @Entity

### @NamedQueries({ @NamedQuery(name = "Student.findByName1", query = "select s from Student s where s.name = ?1"),

### @NamedQuery(name = "Student.findByNameAndRollNumber", query = "select s from Student s where s.name = ?1 and s.rollNumber = ?2"),

### @NamedQuery(name = "Student.findByNameOrRollNumber", query = "select s from Student s where s.name = ?1 or s.rollNumber = ?2") })

### public class Student {

### 

### @Id

### @GeneratedValue(strategy = GenerationType.AUTO)

### private int id;

### 

### @Column(name = "name")

### private String name;

### 

### @Column(name = "roll\_number")

### private String rollNumber;

### 

### @Column(name = "university")

### String university;

### 

### }

### Using @NamedNativeQueries.

### package com.netsurfingzone.entity;

### 

### @Entity

### @NamedNativeQueries({

### @NamedNativeQuery(name = "Student.findByName", query = "select \* from Student where name = ?1", resultClass = Student.class),

### @NamedNativeQuery(name = "Student.findByNameAndRollNumber", query = "select \* from Student where name = ?1 and roll\_number = ?2", resultClass = Student.class),

### @NamedNativeQuery(name = "Student.findByNameOrRollNumber", query = "select \* from Student where name = ?1 or roll\_number = ?2", resultClass = Student.class) })

### public class Student {

### 

### @Id

### @GeneratedValue(strategy = GenerationType.AUTO)

### private int id;

### 

### @Column(name = "name")

### private String name;

### 

### @Column(name = "roll\_number")

### private String rollNumber;

### 

### @Column(name = "university")

### String university;

### 

### }

### Corresponding repository methods.

### @Repository

### public interface StudentRepository extends CrudRepository<Student, Serializable> {

### 

### List<Student> findByName(String name);

### 

### List<Student> findByNameAndRollNumber(String name, String rollNumber);

### 

### List<Student> findByNameOrRollNumber(String name, String rollNumber);

### 

### }

## Difference between findById() and getOne() in Spring Data JPA?

Let’s see some difference between findById() and getOne() methods.

|  |  |
| --- | --- |
| findById() | getOne() |
| 1. The findById() method is available in CrudRepository interface. | 1. The getOne() method is available in JpaRepositpry interface. |
| 2. The findById() method will return null if the record doesn’t exist in the database. | 2. The getOne() method throw EntityNotFoundException if the record doesn’t exist in the database. |
| 3. Internally findById() method use EntityManger find() method. | 3. Internally getOne() method use EntityManger getReference() method. |
| 4. Calling findById() returns a eager fetched entity. | 4. Calling getOne() returns a lazily fetched entity. |

## Difference between delete() vs deleteInBatch() Methods in Spring Data JPA.

### The delete() method has been defined in the CrudRepository interface with the following signature.

### void delete(T entity);

### The deleteInBatch() has been defined in the JpaRepository interface with the following signature.

### void deleteInBatch(Iterable<T> entities);

### The delete() method internally uses EntityManager’s remove() method as below.

### @Transactional

### public void delete(T entity) {

### em.remove(em.contains(entity) ? entity : em.merge(entity));

### }

### The deleteInBatch() internally implemented as below.

### public void deleteInBatch(Iterable<T> entities) {

### applyAndBind(getQueryString(DELETE\_ALL\_QUERY\_STRING, entityInformation.getEntityName()), entities, em)

### .executeUpdate();

### }

### Observe both methods implementation. The first one i.e delete() internally uses remove() method. Before call remove() method it calls contains()/merge() method. Then flow will go inside SessionImpl.java class(fireDelete() method) and a lot of stuff happens.

### On the other hand, deleteInBatch() prepares the query and collect some other information and directly calls the executeUpdate() method.

### -> With the help of delete() method, we can delete a single record at a time whereas using deleteInBatch() we can delete multiple records.

### -> The delete() method is a little slower as compare deleteInBatch() as delete() does some extra stuff than deleteInBatch().

## Difference between deleteAll() Vs deleteAllInBatch() in Spring Data JPA.

### The deleteAll() method is defined in CrudRepository has two overloaded versions.First one – void deleteAll(Iterable<? extends T> entities. Using the above method we can delete all entities which we pass as request data. The deleteAll() internally use delete() method only.

### public void deleteAll(Iterable<? extends T> entities) {

### for (T entity : entities) {

### delete(entity);

### }

### }

### Second one – void deleteAll()-> The above one will delete all records that belong to that repository. The deleteAll() internally uses findAll() and delete() method as below.

### public void deleteAll() {

### for (T element : findAll()) {

### delete(element);

### }

### }

### The query generated in case of deleteAll(). Suppose we are deleting twelve entities.

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### Hibernate: delete from student where id=?

### The deleteAllInBatch() method is defined in the JpaRepository interface as below.

### void deleteAllInBatch()

### Using deleteAllInBatch() method we can delete all entities from database. No need to pass entities as parameters.For example, let’s say we have an entity Student.java and also we have a corresponding repository class called StudentRepository.java something like below.

### package com.netsurfingzone.repository;

### @Repository

### public interface StudentRepository extends JpaRepository<Student, Serializable> {

### }

### If we want to delete all student records we need to do studentRepository.deleteAllInBatch(). See the below sample code.

### @Transactional

### public void deleteAllInBatch() {

### studentRepository.deleteAllInBatch();

### }

### Note – Internally deleteAllInBatch() uses createQuery() method as below.

### public void deleteAllInBatch() {

### em.createQuery(getDeleteAllQueryString()).executeUpdate();

### }

### In case of deleteAllInBatch() the query will generate as below.

### Hibernate: delete from student

### Both deleteAll() and deleteAllInBatch() use to delete all entities. The deleteAll() internally uses CrudRepository delete() method which further uses EntityManger remove() method which perform some other stuff. But in case of deleteAllInBatch(), EntityManger createQuery() method will preapare the query and it directly calls executeUpdate() method.

## How to use Named Query?

### @Query("select s from Student s where s.name = :name")

### List<Student> findByName(@Param("name") String name);

### @Query("select s from Student s where s.name = :name and s.rollNumber = :rollNumber")

### List<Student> findByNameAndRollNumber(@Param("name") String name, @Param("rollNumber") String rollNumber);

## What will happen if we define wrong query methods?

### Suppose we have an entity called Student.java and Student class has fields called id and name. We want all student records for the given name.

### Our query method would be List<Student> findByName(String name);

### Just think by mistake developer writes findName(). instead of findByName(). In this case, while deployment we will get an error.

### org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentController’: Unsatisfied dependency expressed through field ‘studentService’; nested exception is org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentServiceImpl’: Unsatisfied dependency expressed through field ‘studentRepository’; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name ‘studentRepository’: Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: Failed to create query for method public abstract java.util.List com.netsurfingzone.repository.StudentRepository.findName(java.lang.String)! No property findName found for type Student!

### Spring scans all repository methods while deployment itself and checks query methods have been written properly or not.

## How to deine case insensitive search?

### 

### We have to write a case insensitive search query that means for name “john”, our query method should return two records.

### public List<Student> findByNameIgnoreCase(String name);

## Output of below query?

### @Query("select s from Student s where s.roll\_number = ?1")

### List<Student> findStudents(String rollNumber);

### We will get an exception while deployment. Either we need to write JPQL or Native Query.

### Writing a JPQL.

### @Query(“select s from Student s where s.rollNumber= ?1”)

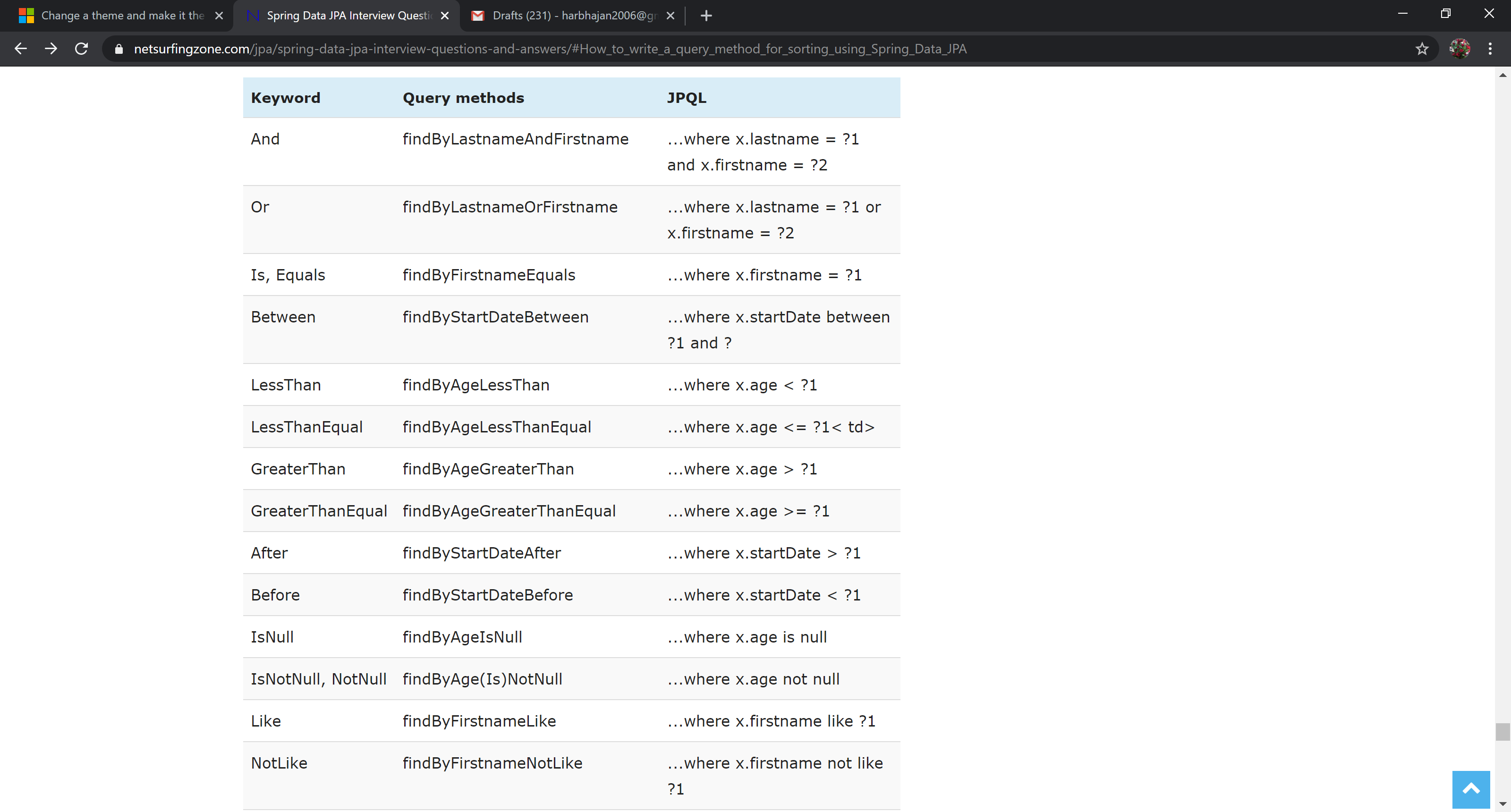
### List<Student> findStudents(String rollNumber);

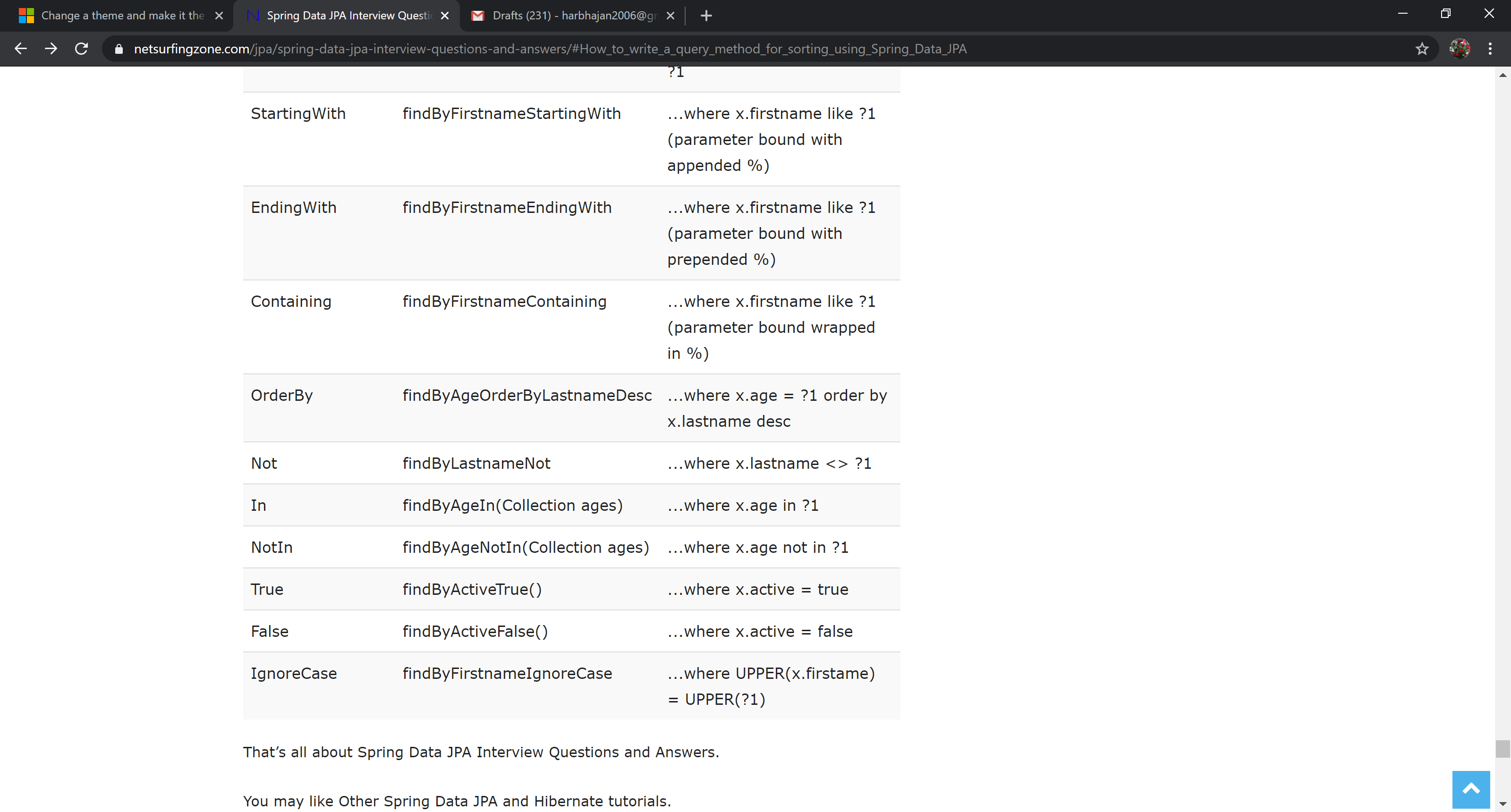
### Writing a native query.

### @Query(value = “select \* from Student s where s.roll\_number = ?1”, nativeQuery = true)

### List<Student> findStudents(String rollNumber);

## List important keyword for QueryMethod





# REFERENCES:

<https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-adding-custom-methods-into-all-repositories/>

<https://www.baeldung.com/spring-data-jpa-pagination-sorting>

<https://www.netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/#How_to_write_a_query_method_for_sorting_using_Spring_Data_JPA>