ReactiveMongo

Sunday, September 26, 2021

10:51 PM

***Custom Queries with Reactive Mongo***

1. ***ReactiveMongoRepository***  interface.

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

***Create a Entity.***

|  |
| --- |
| @Value @AllArgsConstructor @Document(collection = "employees") public class Employee {   @NonFinal @Id String empId;  private String email;  private String password;  private List<Role> roles; } |

*Next step is to declare a repository which extends ReactiveMongoRepository  For the purpose of tutorial, let create an ordinary Employee repository:*

|  |
| --- |
| @Repository public interface EmployeeRepository extends ReactiveMongoRepository<Employee, String> {} |

*This interface will provide us with basic CRUD functionality for Employee entities, including:*

* + *Save*
  + *Retrieve by id*
  + *Update*
  + *Retrieve all*
  + *Remove*

*This, however does not include custom features we may need to do with Employee objects. For instance, we may need to update not a full record , but only specific fields, like password. Same way, we may need to handle changes for nested objects/arrays of the entity. To do this, we can't use derived query methods, but we have to declare a custom repository. Let see how to do this.*

*How to extend ReactiveMongoRepository.*

*The process to extend Spring's reactive Mongo repository with custom functionality takes 3 steps:*

* + *Create an interface that defines custom data management methods, like CustomEmployeeRepository*
  + *Provide an implementation using ReactiveMongoTemplate to access data source. The name of the implementation interface, should have Impl postfix, like CustomEmployeesRepositoryImpl*
  + *Extend the core repository with custom repository interface. For example, EmployeeRepository extends ReactiveMongoRepository, CustomEmployeeRepository*

*Let have a practical example. We already mentioned EmployeeRepository as an entry point for data access operations.*

*We will use this repository in services to handle data source access. However, Spring will create only implementations for CRUD operations, and for custom method updates we will need to provide own implementations. But this is not a rocket science with Spring.*

*The first thing is to create a custom interface to declare our own update methods. Let call it CustomEmployeeRepository:*

|  |
| --- |
| public **interface** CustomEmployeeRepository {   Mono<User> changePassword (String empId, String newPassword);   Mono<User> addNewRole (String empId, Role role);   Mono<User> removeRole (String empId, String permission);   Mono<Boolean> hasPermission (String empId, String permission); } |

*Then, in the step 2, we need to provide a custom implementation.*

*This is the step, where we will need to do most work. We will look on each method separately later, but for now we will provide an essential configuration. What do we need to do:*

* + *Provide a dependency of type ReactiveMongoTemplate. We will use it to access an underlaying data source.*
  + *Add a constructor, where Spring will inject the required bean. Also, annotate the constructor using @Autowired*

*Take a look on the code snippet below:*

|  |
| --- |
| public class CustomEmployeesRepositoryImpl implements CustomEmployeeRepository {   private final ReactiveMongoTemplate mongoTemplate;   @Autowired  public CustomUserRepositoryImpl(ReactiveMongoTemplate mongoTemplate) {  this.mongoTemplate = mongoTemplate;  }   @Override  public Mono<User> changePassword(String userId, String newPassword) {  return null;  }   @Override  public Mono<User> addNewRole(String userId, Role role) {  return null;  }   @Override  public Mono<User> removeRole(String userId, String permission) {  return null;  }   @Override  public Mono<Boolean> hasPermission(String userId, String permission) {  return null;  } } |

*we need to make Spring to know, that we want to use this repository alongside with the built-in one. For this, extend the EmployeeRepository with the CustomEmployeeRepository interface, like this:*

|  |
| --- |
| public interface EmployeeRepository extends ReactiveMongoRepository<User, String>, **CustomEmployeeRepository** {  //... } |

*So, now we are ready to do implementations of custom update methods.*

***Implement custom update functionality***

*In this section we will look how to implement custom updates methods with Reactive MongoDB in Spring. Each method is described separately.*

*Update an entity's field*

*The one of the most common requirements is to change a value of an entity's field. In our example we use confirmed field to store if the employee confirmed her/his account or not. Of course, in our example we concentrate only on database logic, and will not handle a confirmation workflow, so please assume that our****EmployeeService****did everything correctly and now uses a repository to update user's status.*

*Take a look on the code snippet below:*

|  |
| --- |
| @Override public Mono<Employee> changePassword(String empId, String newPassword) {  // step 1  Query query = new Query(Criteria.where("empId").is(empId));    // step 2  Update update = new Update().("password", newPassword);   // step 3 **set** return mongoTemplate.findAndModify(query, update, Employee.class); } |

*Let examine what we do here:*

* + *Create a query object that finds the concrete employee with specific empId*
  + *Create an update object to change the confirmed field's value to true. For this, use set() method that accepts as a first argument field's name (key) and as a second one new value*
  + *Finally execute an update using mongoTemplate object. The method findAndModify is used to apply provided update on documents that match defined criteria.*

*Note, that basically, the flow is same if we use "traditional" synchronous Spring MongoDB.*

*The only difference is the return type - instead of plain User object, reacitve repository returns a Mono.*

***Add to a nested array***

*An another widespread operation you may need to handle in your applications is to add a new entity in a nested array.*

*Let say, that we want to add a new role for the specific Employee. In our example we store permssions in the nested list roles.*

*Again, we don't focus on how actually permission management is done, and assume that EmployeeService does everything as needed. Here is an implementation:*

|  |
| --- |
| @Override public Mono<Employee> addNewRole(String empId, Role role) {  // step 1  Query query = new Query(Criteria.where("empId").is(empId));   // step 2  Update update = new Update().**addToSet**("roles", role);   // step 3  return mongoTemplate.findAndModify(query, update, Employee.class); } |

*Basically, steps are same. The only change is an Update object.*

*We use here****addToSet****method which accepts a name of a nested array - roles and an object to insert in it.*

*There is an another approach to add an object to a nested collection -****push:***

|  |
| --- |
| Update updateWithPush = new Update().**push**("roles", role); |

***Note :***

***The difference between these two methods is following:***

* + *addToSet method does not allow duplicates and inserts an object only if collection does not contain it already.*
  + *push method allows duplicates and can insert the same entity several times*

***Remove from a nested array***

*Like we add something inside a nested collection, we may need to remove it. As we do in our examples role management, we may want to revoke given access rights from the user.*

*To do this we actually need to remove an item from a nested collection. Take a look on the code snippet below:*

|  |
| --- |
| @Override public Mono<Employee> removeRole(String empId, String permission) {  // step 1  Query query = new Query(Criteria.where("empId").is(empId));   // step 2   **Update update = new Update().pull("roles",new BasicDBObject("permission", permission));**   // step 3  return mongoTemplate.findAndModify(query, update, Employee.class); } |

*The update object uses****pull****method to remove a role with a given permission value.*

*We assume that permissions are unique, but in a real life it is better to use some internal unique IDs for the nested object management. We can divide this step into two parts:*

* + *pull* operator accepts as a first argument a name of a nested collection
  + *Second argument is a BasicDBObject which specifies a criteria of which role we want to delete. In our case we use permission field to specify it*

***Query by a nested object's field***

*We already mentioned, that Spring helps us to do querying with derived query methods, like findByName.*

*Although, this will work with object's fields, but what if we want to query by a nested object?*

*For the illustration we declared a method hasPermission which returns a boolean value that indicates if a user has a specific access right, or in terms of our example - does a nested list roles contains an entity with a specific permission value.*

*Let see how we can solve this task:*

|  |
| --- |
| @Override // 1 public Mono<Boolean> hasPermission(String empId, String permission) {   // 2  Query query = new Query(Criteria.where("empId").is(empId))  .addCriteria(Criteria.where("roles.permission").is(permission)); //3   // 4  return mongoTemplate.**exists**(query, Employee.class); } |

*Let go step by step:*

* 1. *Unlike previous methods, this method returns Mono result, which wraps a boolean value that indicates a presence/absense of the desired permission.*
  2. *We need to combine two criterias, because we are looking for the specific user with specific permssion. Query object offers fluent interface and we can chain criterias with addCriteria method.*
  3. *We build a query object which looks for a role entity inside roles array with the needed permission value.*
  4. *Finally, we call exists method, which accepts a query and checks for an existance of the queried data*

*Reference :*[***https://dev.to/iuriimednikov/how-to-build-custom-queries-with-spring-data-reactive-mongodb-1802***](https://dev.to/iuriimednikov/how-to-build-custom-queries-with-spring-data-reactive-mongodb-1802)