**REPOSITORIOS EN SPRING BOOT**

| **Información general** | |
| --- | --- |
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 07 |

| **Información de la Guía** |
| --- |

**OBJETIVO**

Comprender qué son los repositorios en Spring Boot, así como su importancia en el acceso y consulta de datos en un proyecto empresarial.

**CONCEPTOS BÁSICOS**

Bases de datos NoSQL MongoDB, Java, archivos JSON.

**CONTEXTUALIZACIÓN TEÓRICA**

La capa de persistencia tiene como objetivo modelar los datos de un proyecto de software de una manera adecuada y tener acceso a la información que está almacenada en la base de datos. Esta capa depende completamente de los requerimientos del negocio. Ya que son estos últimos los que crean la necesidad de tener tablas o colecciones para almacenar y consultar la información del sistema. Y como ya hemos visto, los documentos son la representación a nivel de objetos en Java de los registros de una base de datos NoSQL.

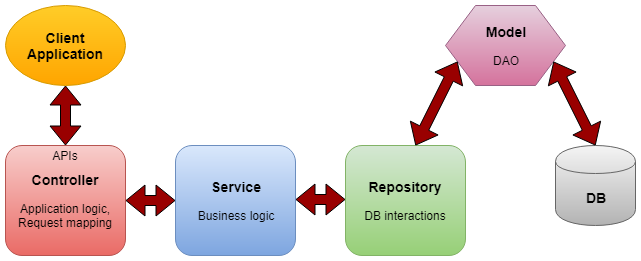
En Spring Boot, el acceso a la base de datos se hace a través de servicios que usan unos repositorios que exponen ciertas funcionalidades de consulta y acceso a datos. Los repositorios representan unas interfaces que extienden de MongoRepository para poder “conectarse” a la base de datos. Estas interfaces gestionan la información, ya sea para buscar, borrar, actualizar o crear un registro en la base de datos. Un repositorio es básicamente un DAO (Data Access Object).

MongoRepository es una interfaz proporcionada por Spring Data MongoDB. Spring Data es un subproyecto de Spring Framework que simplifica la implementación de acceso a datos en diversas tecnologías de persistencia, incluyendo MongoDB.

La interfaz MongoRepository en Spring Data MongoDB proporciona métodos y funcionalidades comunes para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en una colección de MongoDB. Al extender esta interfaz en una interfaz personalizada o en una clase de repositorio, se heredan métodos como save, findById, findAll, delete, etc. Esto permite a los desarrolladores interactuar con la base de datos MongoDB de manera más sencilla y abstraer detalles de implementación.

**Arquitectura del acceso a la información en Spring Boot**

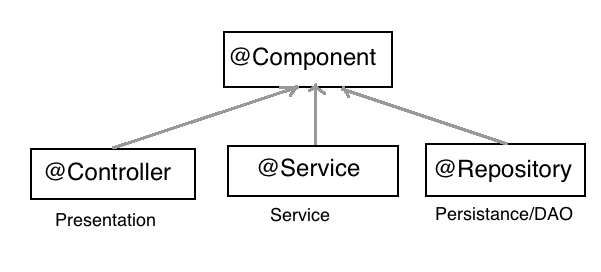
En la siguiente imagen podemos ver una arquitectura muy resumida del acceso a la información desde la capa de presentación (cliente).



El proceso inicia cuando el cliente, que puede ser una aplicación web, móvil, de escritorio, entre otros, envía una solicitud a uno de los controladores o APIs. Estos controladores actúan como intermediarios y requieren servicios que contienen la lógica de negocio necesaria. A su vez, los servicios dependen de repositorios para acceder a los datos almacenados en la base de datos. Estos repositorios, por su parte, se apoyan en el modelo de datos, que comprende las entidades necesarias para interactuar con los registros de la base de datos.

**Componentes en Spring Boot**

En Spring Boot se suele usar la anotación @Component para indicar que hay ciertos objetos que se deben inicializar automáticamente y se le debe gestionar la memoria de acuerdo a su propósito, por ejemplo: controladores, servicios, repositorios, etc.

****

Podemos definirlos así:

* @Component: Esta es una anotación de estereotipo de propósito general que indica que la clase es un componente de spring.
* @Repository: Esto es para indicar que la clase define un depósito de datos.
* @Controller: Indica que una clase en particular cumple la función de un controlador. Actúa como un estereotipo para la clase anotada, lo que indica su función.
* @Service: mantiene la lógica de negocio y los métodos de llamada en la capa de repository.

Por ahora, en la capa de persistencia nos centraremos en los repositorios.

**PRECAUCIONES Y RECOMENDACIONES**

Recuerde verificar que tiene instalado el JDK de Java (preferiblemente la versión 21). Asegúrese de que tiene el servidor de MongoDB (mongod) en ejecución.

**ARTEFACTOS**

Se requiere tener instalado IntelliJ IDEA en su versión Ultimate, así como MongoDB y el controlador de versiones GIT.

**EVALUACIÓN O RESULTADO**

Se espera que el estudiante logre crear repositorios para acceder a la base de datos, tanto para guardar registros como modificarlos, eliminarlos y consultarlos.

| **Procedimiento** |
| --- |

1. En esta guía, continuaremos con el ejemplo de la tienda que hemos estado desarrollando en la guía anterior.
2. Cree el paquete co.edu.uniquindio.proyecto.repositorios en la carpeta main/java.
3. Dentro del paquete creado en el punto anterior, cree un interface que se llame ClienteRepo, este interface debe extender de MongoRepository. MongoRepository es genérico, por lo tanto debemos definir primero el tipo de dato del documento al que referencia el repositorio y segundo debe definir el tipo de dato de su @Id, para nuestro caso será Cliente y ObjectId respectivamente. Tenga en cuenta que por cada @Document será necesario crear un repositorio.

La interface ClienteRepo debe tener la anotación @Repository.

| package co.edu.uniquindio.proyecto.repositorios;  import co.edu.uniquindio.proyecto.modelo.Cliente;  import org.bson.types.ObjectId;  import org.springframework.data.mongodb.repository.MongoRepository;  import org.springframework.stereotype.Repository;  @Repository  public interface ClienteRepo extends MongoRepository<Cliente, ObjectId> {  } |
| --- |

**NOTA:** Cambie proyecto por el nombre de su proyecto (o el nombre del paquete que ya haya creado previamente (así mismo con el resto de la guía).

1. El repositorio actúa como un componente en Spring Boot que tiene acceso directo a la colección correspondiente. Esto nos permite utilizarlo para crear, eliminar, modificar o consultar clientes en la base de datos de MongoDB.

Debido a que el ClienteRepo extiende de MongoRepository entonces tenemos unos métodos listos y completamente funcionales. Ejemplo:

| save(objeto) | Si el registro es nuevo lo guarda en la colección de la base de datos. Si ya existe un registro con el mismo id entonces lo actualiza. |
| --- | --- |
| delete(objeto) | Elimina un registro que esté en la base de datos. |
| deleteById(id) | Elimina un registro que esté en la base de datos dado el id. |
| findAll() | Retorna una lista con todos los registros que están almacenados en una colección en la base de datos. |
| findById(id) | Retorna un registro que está en la base de datos dado su id. |
| count() | Devuelve la cantidad de registros que tiene una colección. |
| exists(objeto) | Retorna un boolean indicando si el objeto existe en la base de datos. |
| existsById(id) | Retorna un boolean indicando si el objeto existe en la base de datos dado el id. |

De igual manera, si se necesita un consulta que no esté dentro de las que ya dispone MongoRepository, entonces dicha consulta deberá ser programada dentro de los repositorios de cada entidad. Estas consultas las iremos estudiando en futuras guías.

1. Por ejemplo, el repositorio para los productos quedaría así (asumiendo que la llave primaria del documento Producto es de tipo ObjectId):

| package co.edu.uniquindio.proyecto.repositorios;  import co.edu.uniquindio.proyecto.modelo.Producto;  import org.bson.types.ObjectId;  import org.springframework.data.mongodb.repository.MongoRepository;  import org.springframework.stereotype.Repository;  @Repository  public interface ProductoRepo extends MongoRepository<Producto, ObjectId> {  } |
| --- |

1. Programe los repositorios para todas aquellas clases marcadas con la anotación @Document. Debe seguir la misma estructura usada para crear el repositorio del cliente y del producto.
2. Para probar los métodos de los repositorios crearemos una clase de test. Cree un paquete en la carpeta test/java que se llame co.edu.uniquindio.proyecto.test. Luego, cree una clase que se llame ClienteTest. Agregue lo siguiente:

| package co.edu.uniquindio.proyecto.test;  import co.edu.uniquindio.proyecto.repositorios.ClienteRepo;  import org.junit.jupiter.api.Test;  import org.bson.types.ObjectId;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import static org.junit.jupiter.api.Assertions.\*;  @SpringBootTest  public class ClienteTest {  @Autowired  private ClienteRepo clienteRepo;  } |
| --- |

La clase ClienteTest necesita la anotación @SpringBootTest para que se ejecute en el contexto de prueba. Además, necesitamos el repositorio que creamos anteriormente para poder tener acceso a las funcionalidades básicas de registro, eliminación, actualización y lectura (CRUD). La variable que representa el repositorio necesita de la anotación @Autowired para que sea inicializada automáticamente por Spring.

1. Ahora agregue cuatro métodos para verificar el CRUD para la colección de clientes. Primero vamos a programar el método para probar la creación de un nuevo cliente:

| @Test  public void registrarTest(){  //Creamos un cliente con sus propiedades  Cliente cliente = Cliente.*builder*()  .cedula("123456")  .nombre("Pepito")  .email("pepito@email.com")  .telefonos( List.*of*("telefono1", "telefono2") ).build();  //Guardamos el cliente en la base de datos  Cliente clienteCreado = clienteRepo.save(cliente);  //Verificamos que se haya guardado validando que no sea null  *assertNotNull*(clienteCreado);  } |
| --- |

Para que esto funcione, asegúrese de tener en la clase Cliente un constructor con todos los atributos excepto el código (\_id) y marque dicho constructor con la anotación de Lombok @Builder.

La línea assertNotNull(registro) se utiliza para verificar que el objeto clienteCreado no sea nulo durante la ejecución de una prueba unitaria en JUnit. Si el objeto es nulo, la prueba fallará, lo que indica que algo inesperado ocurrió en la lógica que estamos probando.

1. Ejecute el método y verifique los resultados. En MongoDB Compass, diríjase a la base de datos y a la colección donde se debería haber guardado el cliente para confirmar que se registró correctamente.
2. Para probar la actualización, programaremos el siguiente método:

| @Test  public void actualizarTest(){  //Definimos el id del cliente (de MongoDB)  ObjectId id = new ObjectId("XXXXXXX");  //Obtenemos el cliente con el id XXXXXXX  Cliente cliente = clienteRepo.findById(id).orElseThrow();  //Modificar el email del cliente  ciente.setEmail("nuevoemail@email.com");  //Guardamos nuevamente el cliente  clienteRepo.save( cliente );  //Obtenemos el cliente con el id XXXXXXX nuevamente  Cliente clienteActualizado = clienteRepo.findById(id).orElseThrow();;  //Verificamos que el email se haya actualizado  *assertEquals*("nuevoemail@email.com", clienteActualizado.getEmail());  } |
| --- |

Reemplace el id XXXXXXX por el \_id con el que haya quedado guardado el cliente al haber ejecutado el test de registro (el \_id lo conocemos al hacer el punto anterior y corroborar el registro desde MongoDB Compass).

La instrucción assertEquals se utiliza para verificar que el valor del correo electrónico (getEmail()) en el objeto clienteActualizado es igual a la cadena "nuevoemail@email.com" durante la ejecución de una prueba unitaria. Si estos valores no son iguales, la prueba fallará.

1. Para verificar los resultados, ejecute el método actualizarTest(). Luego, en MongoDB Compass, acceda a la base de datos y la colección donde se debería haber actualizado el cliente para confirmar que los cambios se aplicaron correctamente.
2. Para probar el método que devuelve la lista de todos los clientes, programaremos el siguiente método:

| @Test  public void listarTodosTest(){  //Obtenemos la lista de todos los clientes (por ahora solo tenemos 1)  List<Cliente> lista = clienteRepo.findAll();  //Imprimimos los clientes, se hace uso de una función lambda  lista.forEach(System.*out*::println);  //Verificamos que solo exista un cliente  *assertEquals*(1, lista.size());  } |
| --- |

1. Ejecute el método y verifique los resultados. Agregue otro cliente usando el método de test de registro y cambie el 1 del assertEquals por un 2 y verifique que todo funciona bien. ¿Qué pasa si no se hace el cambio en el assertEquals y se deja el 1?
2. Para probar la eliminación, programaremos el siguiente método:

| @Test  public void eliminarTest(){  //Definimos el id del cliente que queremos borrar  ObjectId id = new ObjectId("XXXXXXX");  //Borramos el cliente con el id XXXXXXX  clienteRepo.deleteById(id);  //Obtenemos el cliente con el id XXXXXXX  Cliente cliente = clienteRepo.findById(id).orElse(null);  //Verificamos que el cliente no exista, debe ser null ya que fue eliminado  *assertNull*(cliente);  } |
| --- |

Reemplace el id XXXXXXX por el \_id con el que haya quedado guardado el cliente al haber ejecutado el test de registro (el \_id lo conocemos al hacer el punto 8 y corroborar el registro desde MongoDB Compass).

1. Para verificar los resultados, ejecute el método eliminarTest(). Luego, en MongoDB Compass, navegue hasta la base de datos y la colección donde se encontraba el cliente para confirmar que se eliminó correctamente.
2. Programe la clase ProductoTest y TransaccionTest y pruebe el CRUD tal como se hizo con la colección de clientes.

**NOTA:** Tenga en cuenta que las pruebas a los métodos de los repositorios serán reemplazadas más adelante por las pruebas de los servicios de negocio que representan los requerimientos del proyecto final.

1. Recuerde hacer commit en su repositorio de git con todo lo realizado en esta guía.

| **Para la próxima clase** |
| --- |

* De ahora en adelante, se trabajará con el modelo de clases del proyecto final. Por lo tanto, es necesario crear las clases que se mapearán como documentos en la base de datos, incluyendo sus respectivas propiedades. Además, debe implementar los repositorios correspondientes para cada documento.
* Leer este post: <https://www.baeldung.com/queries-in-spring-data-mongodb>
* Investigue sobre la inferencia de queries en Spring Boot MongoDB, a qué se refiere y para qué sirve.