**SERVICIOS DE NEGOCIO EN SPRING BOOT**

| **Información general** | |
| --- | --- |
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 05 |

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

**OBJETIVO**

Comprender qué son los servicios en Spring Boot, así como su importancia en la capa de lógica de negocio para el acceso y consulta de datos en un proyecto empresarial.

**CONCEPTOS BÁSICOS**

Bases de datos NoSQL MongoDB, Java, archivos JSON.

**CONTEXTUALIZACIÓN TEÓRICA**

Los servicios de negocio se refieren a componentes o módulos que encapsulan la lógica empresarial de una aplicación. Estos servicios son responsables de implementar y gestionar las reglas y procesos de negocio que son específicos de la aplicación.

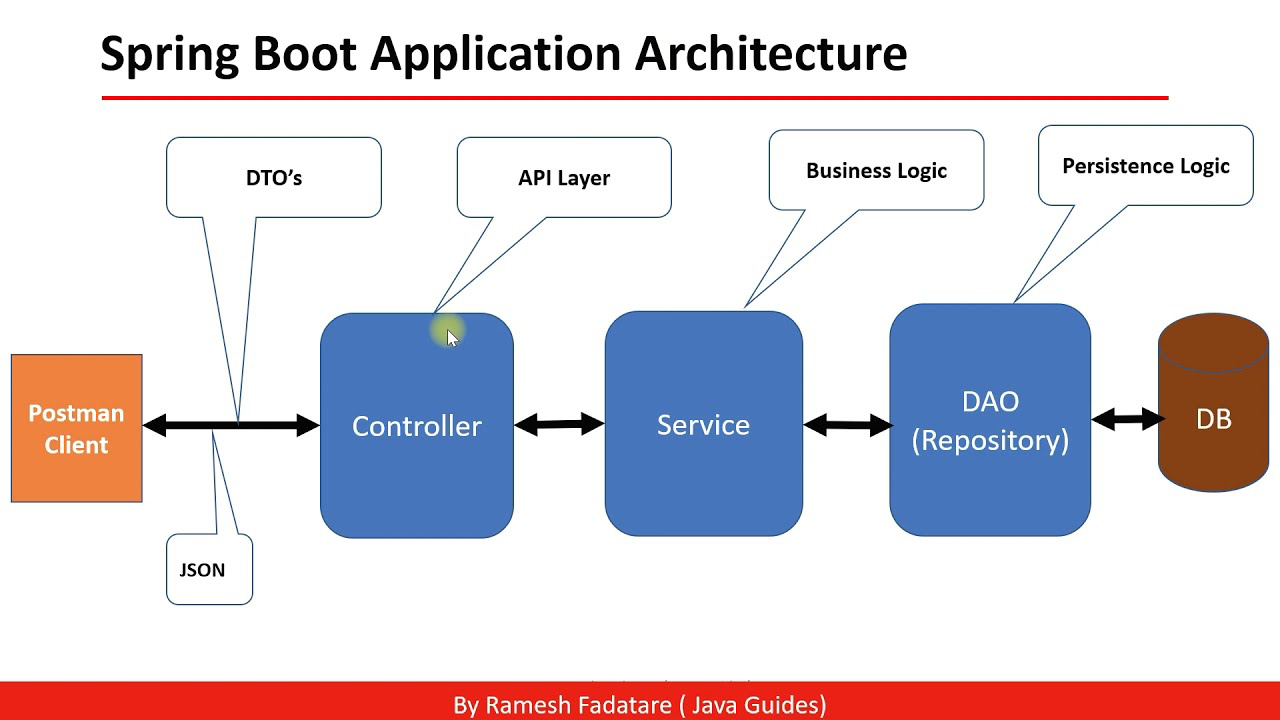
En Spring Boot, la anotación @Service (que hace referencia a un @Component) se utiliza para marcar una clase como un componente de servicio en la capa de servicios de la arquitectura de aplicación. Este tipo de clases generalmente contienen la lógica de negocio de la aplicación y son responsables de realizar operaciones específicas del dominio.

Al utilizar la anotación @Service, estamos indicando a Spring que la clase debe ser considerada como un componente de servicio y se gestionará como tal dentro del contexto de la aplicación. Esto significa que la clase marcada con @Service se registrará automáticamente como un bean de Spring y se podrá inyectar en otras clases donde sea necesario.

Aquí hay algunas características clave de los servicios de negocio en el contexto de Spring Boot:

* Lógica de Negocio: Los servicios de negocio encapsulan la lógica empresarial de la aplicación, implementando operaciones, validaciones y reglas específicas del dominio.
* Reutilización: Los servicios de negocio pueden ser diseñados para ser reutilizables en diferentes partes de la aplicación. Esto promueve la modularidad y facilita la gestión de la lógica de negocio.
* Separación de Responsabilidades: Spring Boot sigue el principio de separación de responsabilidades, por lo que los servicios de negocio se encargan de la lógica empresarial mientras que otros componentes, como los controladores y las capas de persistencia, se centran en tareas específicas.
* Inyección de Dependencias: Spring Boot utiliza la inversión de control (IoC) y la inyección de dependencias para gestionar la creación y administración de componentes. Los servicios de negocio pueden beneficiarse de la inyección de dependencias para obtener fácilmente otras dependencias necesarias.
* Transacciones: En aplicaciones empresariales, a menudo es necesario gestionar transacciones. Spring Boot ofrece soporte para la gestión de transacciones, lo que permite que los servicios de negocio participen en transacciones de base de datos de manera coherente.

La siguiente imagen ilustra el proceso de comunicación entre las diferentes capas de la arquitectura de una aplicación de Spring Boot

****

El proceso de petición desde el cliente a los servicios de Spring Boot sigue el modelo de arquitectura Cliente-Servidor y se basa en el protocolo HTTP (o HTTPS). Aquí hay un resumen general del proceso:

* Un cliente (como un navegador web o una aplicación móvil) inicia una solicitud enviando una petición HTTP al servidor Spring Boot.
* En el servidor Spring Boot, las peticiones son manejadas por controladores. Los controladores están anotados con @Controller, @RestController u otras anotaciones específicas de Spring.
* El controlador realiza las operaciones necesarias en función de la lógica de negocios. Puede interactuar con servicios, acceder a bases de datos, realizar operaciones y preparar datos para enviar como respuesta.
* Generación de la respuesta: Una vez que el controlador ha completado su lógica, genera una respuesta. La respuesta generada por el controlador se envía de vuelta al cliente a través del protocolo HTTP.
* El cliente (por ejemplo, un navegador web o una aplicación móvil) recibe la respuesta y realiza acciones según la lógica de la aplicación.

**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 programar los servicios de negocio que den solución a los requerimientos funcionales y no funcionales de un proyecto empresarial en Spring Boot.

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

1. En Spring Boot, un servicio requiere de uno o varios repositorios. Un Servicio está compuesto por una interface donde se definen los requerimientos de negocio para cada entidad (o para cada rol) y una clase donde se implementa dicha interface y se programa la lógica de negocio, con las validaciones necesarias.

Cree un paquete que se llame co.edu.uniquindio.proyecto.servicios en la carpeta main/java.

**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. En el paquete creado en el punto anterior cree un interface que se llame UsuarioServicio y defina los métodos que representen los servicios de negocio asociados a la colección de los usuarios que sean pertinentes para el proyecto final.

Para esta guía se implementará un CRUD a manera de ejemplo, por lo tanto, se necesitan los siguientes métodos (para el proyecto final debe agregar más métodos):

| package co.edu.uniquindio.proyecto.servicios;  import co.edu.uniquindio.proyecto.dto.CrearUsuarioDTO;  import co.edu.uniquindio.proyecto.dto.EditarUsuarioDTO;  import co.edu.uniquindio.proyecto.dto.UsuarioDTO;  import java.util.List;  public interface UsuarioServicio {  void crear(CrearUsuarioDTO crearUsuarioDTO) throws Exception;  void eliminar(String id) throws Exception;  void editar(EditarUsuarioDTO editarUsuarioDTO) throws Exception;  UsuarioDTO obtener(String id) throws Exception;  List<UsuarioDTO> listarTodos(String nombre, String ciudad);  } |
| --- |

Como puede ver, algunos métodos tienen un throws Exception, esto es debido a que siempre debemos validar y capturar las excepciones que puedan aparecer al momento de manipular la base de datos (consultar, insertar, actualizar o eliminar), esto se conoce como programación defensiva.

Para más información:

* <https://latteandcode.medium.com/qu%C3%A9-es-eso-de-la-programaci%C3%B3n-defensiva-6a89d3bcab8e>

**NOTA:** Es una buena práctica primero definir los servicios por medio de interfaces ya que allí podemos listar todos los requisitos, sus entradas de datos y sus retornos. Básicamente en el interface definimos qué hay que hacer pero no el cómo, el cómo lo haremos más adelante cuando se programen las clases que implementen dichos interfaces.

1. Usar las clases de persistencia (las clases anotadas con @Document o @Entity) directamente en la capa de negocio de una aplicación Spring Boot puede llevar a problemas potenciales en ciertos casos. Aquí hay algunas razones por las cuales usar entidades en la capa de negocio podría considerarse problemático:

* Exposición de detalles de implementación: Las entidades suelen estar fuertemente relacionadas con la estructura de la base de datos. Usarlas directamente en la capa de negocio puede exponer detalles de implementación que no deberían ser visibles en esa capa. Esto puede hacer que el código de negocio sea más frágil y menos mantenible, ya que los cambios en la base de datos pueden tener un impacto directo en la lógica de negocio.
* Problemas de acoplamiento: Utilizar entidades directamente puede llevar a un alto acoplamiento entre la capa de negocio y la capa de acceso a datos (por ejemplo, el repositorio JPA). Esto dificulta la evolución y el mantenimiento de la aplicación, ya que cualquier cambio en la capa de datos podría requerir cambios en la capa de negocio.
* Falta de flexibilidad: Las entidades pueden estar diseñadas para satisfacer las necesidades de la base de datos, lo que significa que pueden contener propiedades que no son relevantes para la lógica de negocio. Esto puede llevar a un exceso de carga de datos y a un rendimiento deficiente en operaciones que no necesitan todos los campos de la entidad.
* Dificultad en la gestión de transacciones: En algunos casos, usar entidades directamente puede complicar la gestión de transacciones en la capa de negocio, especialmente si se realizan operaciones complejas que involucran varias entidades y relaciones.

Para abordar estos problemas, es común utilizar DTOs (Data Transfer Objects) o clases específicas de dominio en la capa de negocio en lugar de entidades directamente. Estas clases se diseñan para satisfacer las necesidades específicas de la lógica de negocio y pueden proporcionar un nivel de abstracción entre la capa de negocio y la capa de acceso a datos. Esto permite que la capa de negocio tenga su propia representación de los datos que puede ser más coherente con las necesidades del negocio.

Para conocer un poco más sobre cada solución podemos visitar:

* + <https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application>
  + <https://www.arquitecturajava.com/data-transfer-object-dto-un-concepto-clave/>
  + <https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dto>

Como se menciona en este punto, por buenas prácticas, las clases anotadas con @Entity o @Document no deberían usarse como parámetros en los métodos de negocio.

1. Cree un paquete que se llame co.edu.uniquindio.proyecto.servicios.impl. A dicho paquete añádale una clase que se llame UsuarioServicioImpl e implemente allí el interface UsuarioServicio creado previamente, así:

| package co.edu.uniquindio.proyecto.servicios.impl;  import co.edu.uniquindio.proyecto.repositorios.UsuarioRepo;  import co.edu.uniquindio.proyecto.servicios.UsuarioServicio;  import org.springframework.stereotype.Service;  @Service  public class UsuarioServicioImpl implements UsuarioServicio {  private final UsuarioRepo usuarioRepo;  public UsuarioServicioImpl(UsuarioRepo usuarioRepo) {  this.usuarioRepo = usuarioRepo;  }  } |
| --- |

Esta clase debe tener la anotación @Service para que Spring Boot la inicialice correctamente. Además, necesitamos declarar una variable que represente el repositorio (UsuarioRepo) para poder invocar sus métodos de acceso a la base de datos. Esta variable debe ser final. Para inicializar este atributo se usa el constructor de la clase (Spring Boot se encarga de instanciar esta clase cuando sea necesaria) o la anotación @Autowired.

1. Tenga en cuenta que en lugar de hacer el constructor de forma manual puede usar la anotación: @RequiredArgsConstructor de lombok. Esta anotación crea automáticamente un constructor con los atributos de la clase que sean final. Por lo tanto, la clase anterior puede quedar así:

| package co.edu.uniquindio.proyecto.servicios.impl;  import co.edu.uniquindio.proyecto.repositorios.UsuarioRepo;  import co.edu.uniquindio.proyecto.servicios.UsuarioServicio;  import lombok.RequiredArgsConstructor;  import org.springframework.stereotype.Service;  @Service  @RequiredArgsConstructor  public class UsuarioServicioImpl implements UsuarioServicio {  private final UsuarioRepo usuarioRepo;  } |
| --- |

1. Implemente todos los métodos del interface, por ejemplo, el método de registro se debe ver así:

| @Override  public void crear(CrearUsuarioDTO crearUsuarioDTO) throws Exception {    //Mapeamos (pasamos) los datos del DTO a un objeto de tipo Usuario  Usuario usuario = new Usuario();  usuario.setNombre(crearUsuarioDTO.nombre());  usuario.setCiudad(crearUsuarioDTO.ciudad());  usuario.setDireccion(crearUsuarioDTO.direccion());  usuario.setEmail(crearUsuarioDTO.email());  usuario.setPassword(crearUsuarioDTO.password());  usuario.setTelefono(crearUsuarioDTO.telefono());  usuario.setEstado(EstadoUsuario.*INACTIVO*);  usuario.setRol(Rol.*CLIENTE*);  usuario.setFechaRegistro(LocalDateTime.*now*());  //Guardamos el usuario en la base de datos  usuarioRepo.save(usuario);  } |
| --- |

Aunque en el parámetro del método llega un DTO, dicho objeto se debe convertir (mapear) a un objeto de tipo Usuario ya que a nivel de base de datos (para guardar, actualizar, etc.) estas operaciones se hacen con la clase anotada con @Document.

**NOTA:** El mapeo de un DTO a una clase de tipo Document (y viceversa) también se puede realizar por medio de librerías que automatizan este proceso, por ejemplo, la más popular es MapStruct: <https://mapstruct.org/documentation/installation/>.

1. Agregue las siguientes dependencias al archivo build.gradle.kts.

| *implementation*("org.mapstruct:mapstruct:1.6.3")  *annotationProcessor*("org.mapstruct:mapstruct-processor:1.6.3") |
| --- |

Sincronice y descargue las dependencias por medio de Gradle.

1. Cree la interface UsuarioMapper dentro del paquete co.edu.uniquindio.proyecto.mapper y agregue lo siguiente:

| package co.edu.uniquindio.proyecto.mapper;  import co.edu.uniquindio.proyecto.dto.CrearUsuarioDTO;  import co.edu.uniquindio.proyecto.dto.UsuarioDTO;  import co.edu.uniquindio.proyecto.modelo.documentos.Usuario;  import org.bson.types.ObjectId;  import org.mapstruct.Mapper;  import org.mapstruct.Mapping;  @Mapper(componentModel = "spring")  public interface UsuarioMapper {  @Mapping(target = "rol", constant = "CLIENTE")  @Mapping(target = "estado", constant = "INACTIVO")  @Mapping(target = "fechaRegistro", expression = "java(java.time.LocalDateTime.now())")  Usuario toDocument(CrearUsuarioDTO usuarioDTO);  UsuarioDTO toDTO(Usuario usuario);  // Metodo para mapear de ObjectId a String  default String map(ObjectId value) {  return value != null ? value.toString() : null;  }  } |
| --- |

**IMPORTANTE:** Asegúrese que los nombres de los atributos que comparten los DTO y las clases del modelo coincidan. Por ejemplo, la clase Usuario tiene el campo nombre, email, password, telefono, y CrearUsuarioDTO también los tiene y se llaman igual.

Para que el Mapper funcione correctamente, se debe generar un constructor con todos los atributos omitiendo el id en la clase Usuario.

Para más información: <https://chatgpt.com/share/67d8822b-7828-800e-a9fa-7f94e003a73d>

1. Ahora, modifique el método que hicimos en el punto 6 de esta guía para que quede así:

| @Override  public void crear(CrearUsuarioDTO crearUsuarioDTO) throws Exception {  Usuario usuario = usuarioMapper.toDocument(crearUsuarioDTO);  usuarioRepo.save(usuario);  } |
| --- |

Dado que estamos usando usuarioMapper para hacer el mapeo de los datos, debe agregar la siguiente variable a nivel de la clase:

| private final UsuarioMapper usuarioMapper; |
| --- |

Observe que la implementación queda más limpia y los mapeos son hechos directamente por UsuarioMapper.

1. Al momento de crear (o registrar) a un nuevo usuario, se debe validar que su email no esté en uso, ya que este campo debería ser único. Para esto, antes de crear el objeto usuario en el método anterior, agregue lo siguiente:

| if( existeEmail(crearUsuarioDTO.email()) ){  throw new Exception("El correo "+crearUsuarioDTO.email()+" ya está en uso");  } |
| --- |

El manejo de excepciones es muy útil porque nos permite tener un control sobre el posible error que vaya a aparecer, evitamos que los errores sean producidos desde el motor de base de datos y además, podemos generar mensajes personalizados para las excepciones (esto es útil para la GUI).

**NOTA:** Se recomienda usar excepciones propias, cree una excepción para usarla cuando haya datos repetidos. Recuerde agregar esta excepción a la clase RestExceptionHandler y use el código de respuesta 409.

1. Programe el método existeEmail() en la clase UsuarioServicioImpl para que quede así:

| private boolean existeEmail(String email) {  return usuarioRepo.findByEmail(email).isPresent();  } |
| --- |

Dentro del interface UsuarioRepo agregue el método findByEmail() así:

| Optional<Usuario> findByEmail(String email); |
| --- |

Tenga en cuenta que findBy es un nombre propio de métodos de Spring Data MongoDB que puede inferir la consulta. En este caso se asume que el método findByEmail() buscará un usuario con el email que llega por parámetro. Si lo encuentra, dentro del objeto Optional estará presente dicho usuario, sino, el Optional estará vacío. Es importante indicar que “email” debe ser un nombre de un atributo de la clase Usuario.

Para más información sobre el uso de findBy() se recomienda visitar:

* <https://bushansirgur.in/spring-data-jpa-finder-methods-by-field-name-with-examples/>

1. Las demás implementaciones pueden quedar así:

Para el método de editar, primero buscamos al Usuario y le asignamos los campos que se pueden cambiar, en este caso los datos que tiene EditarUsuarioDTO.

| @Override  public void editar(EditarUsuarioDTO editarUsuarioDTO) throws Exception {  //Validamos el id  if (!ObjectId.*isValid*(editarUsuarioDTO.id())) {  throw new Exception("No se encontró el usuario con el id "+editarUsuarioDTO.id());  }  //Buscamos el usuario que se quiere actualizar  ObjectId objectId = new ObjectId(editarUsuarioDTO.id());  Optional<Usuario> usuarioOptional = usuarioRepo.findById(objectId);  //Si no se encontró el usuario, lanzamos una excepción  if(usuarioOptional.isEmpty()){  throw new Exception("No se encontró el usuario con el id "+editarUsuarioDTO.id());  }  // Mapear los datos actualizados al usuario existente  Usuario usuario = usuarioOptional.get();  usuarioMapper.toDocument(editarUsuarioDTO, usuario);  //Como el objeto usuario ya tiene un id, el save() no crea un nuevo registro sino que actualiza el que ya existe  usuarioRepo.save(usuarioModificado);  } |
| --- |

Es necesario agregar el siguiente método en la interface UsuarioMapper:

| @Mapping(target = "id", ignore = true)  @Mapping(target = "estado", ignore = true)  @Mapping(target = "rol", ignore = true)  @Mapping(target = "fechaRegistro", ignore = true)  @Mapping(target = "codigoValidacion", ignore = true)  @Mapping(target = "email", ignore = true)  @Mapping(target = "password", ignore = true)  void toDocument(EditarUsuarioDTO editarUsuarioDTO, @MappingTarget Usuario usuario); |
| --- |

Para más información: <https://chatgpt.com/share/67d8838b-ea4c-800e-b1bf-4b82fb64763a>

1. Para el método de obtener, buscamos el Usuario y lo pasamos a un UsuarioDTO.

| @Override  public UsuarioDTO obtener(String id) throws Exception {  //Validamos el id  if (!ObjectId.*isValid*(id)) {  throw new Exception("No se encontró el usuario con el id "+id);  }  //Buscamos el usuario que se quiere obtener  ObjectId objectId = new ObjectId(id);  Optional<Usuario> usuarioOptional = usuarioRepo.findById(objectId);  //Si no se encontró el usuario, lanzamos una excepción  if(usuarioOptional.isEmpty()){  throw new Exception("No se encontró el usuario con el id "+id);  }  //Retornamos el usuario encontrado convertido a DTO  return usuarioMapper.toDTO(usuarioOptional.get());  } |
| --- |

1. Para el método de eliminar, buscamos el Usuario y le cambiamos su estado a ELIMINADO.

| @Override  public void eliminar(String id) throws Exception {  //Validamos el id  if (!ObjectId.*isValid*(id)) {  throw new Exception("No se encontró el usuario con el id "+id);  }  //Buscamos el usuario que se quiere obtener  ObjectId objectId = new ObjectId(id);  Optional<Usuario> usuarioOptional = usuarioRepo.findById(objectId);  //Si no se encontró el usuario, lanzamos una excepción  if(usuarioOptional.isEmpty()){  throw new Exception("No se encontró el usuario con el id "+id);  }  //Obtenemos el usuario que se quiere eliminar y le asignamos el estado eliminado  Usuario usuario = usuarioOptional.get();  usuario.setEstado(EstadoUsuario.*ELIMINADO*);  //Como el objeto usuario ya tiene un id, el save() no crea un nuevo registro sino que actualiza el que ya existe  usuarioRepo.save(usuario);  } |
| --- |

Finalmente, para el método de listar, obtenemos todos los registros de la colección de usuarios y los transformamos a UsuarioDTO.

| @Override  public List<UsuarioDTO> listarTodos(String nombre, String ciudad) {  List<Usuario> usuarios = usuarioRepo.findAll();  return usuarios.stream()  .map(usuarioMapper::toDTO)  .collect(Collectors.*toList*());  } |
| --- |

**NOTA 2:** Piense en un método de apoyo para no tener que repetir la sección de código que busca un usuario dado su id, cuando no existe lanza la excepción, pero si existe lo guarda en una variable (observe que el método actualizar, eliminar y obtener información del usuario lo tienen).

1. En lugar de lanzar Exception cuando no se encuentra un elemento, mejor cree una excepción nueva para que maneje estos errores. Recuerde agregar esta excepción a la clase RestExceptionHandler y use el código de respuesta 404.
2. Pruebe los métodos creados previamente usando la clase UsuarioControladorTest creada en guías anteriores. Verifique que todo funcione correctamente y que los datos se vean reflejados en la base de datos de MongoDB. Recuerde que MongoDB debe estar en ejecución.
3. Modifique el método listarTodos() de la clase UsuarioServicioImpl para que haga uso de un paginador y los filtros que llegan por parámetro, así:

| @Override  public List<UsuarioDTO> listarTodos(String nombre, String ciudad, int pagina) {  // Validar que la página no sea menor a 0  if(pagina < 0) throw new RuntimeException("La página no puede ser menor a 0");  // Crear criterios dinámicos  Criteria criteria = new Criteria();  if (nombre != null && !nombre.isEmpty()) {  criteria.and("nombre").regex(nombre, "i"); // Ignora a mayúsculas/minúsculas  }  if (ciudad != null && !ciudad.isEmpty()) {  criteria.and("ciudad").regex(ciudad, "i");  }  // Crear la consulta con los criterios y la paginación de 5 elementos por página  Query query = new Query(criteria).with(PageRequest.*of*(pagina, 5));  List<Usuario> usuarios = mongoTemplate.find(query, Usuario.class);  // Convertir la lista de usuarios a una lista de DTOs  return usuarios.stream()  .map(usuarioMapper::toDTO)  .toList();  } |
| --- |

Criteria es una clase de Spring Data MongoDB que permite construir consultas dinámicas para MongoDB de una manera más programática y flexible. Se usa junto con Query y MongoTemplate para definir condiciones de búsqueda en la base de datos. Esta es otra forma que tenemos disponible para escribir consultas. Para poder usarlo, debe crear la siguiente variable en la clase UsuarioServicioImpl:

| private final MongoTemplate mongoTemplate; |
| --- |

Y debe agregar los siguientes imports:

| import org.springframework.data.mongodb.core.MongoTemplate;  import org.springframework.data.mongodb.core.query.Criteria;  import org.springframework.data.mongodb.core.query.Query;  import org.springframework.data.domain.PageRequest; |
| --- |

Para más información:

* <https://www.baeldung.com/spring-data-mongo-several-criteria>
* <https://docs.spring.io/spring-data/mongodb/reference/mongodb/template-query-operations.html>
* <https://chatgpt.com/share/67da1357-a8b0-800e-bd8d-eedd5213b0bb>

1. Pruebe el método de listar usuarios que modificamos en el punto anterior. Envíe el parámetro de la página y opcionalmente la ciudad o el nombre.

Por ejemplo: GET /api/usuarios?pagina=0&ciudad=Armenia

1. Dado que el proyecto requiere de notificaciones en tiempo real, se requiere hacer uso de WebSockets, para esto, agregue la siguiente dependencia a su build.gradle.kts:

| *implementation*("org.springframework.boot:spring-boot-starter-websocket") |
| --- |

Sincronice y descargue las dependencias por medio de Gradle.

1. Cree la clase WebSocketConfig en el paquete co.edu.uniquindio.proyecto.config así:

| package co.edu.uniquindio.proyecto.config;  import org.springframework.context.annotation.Configuration;  import org.springframework.messaging.simp.config.MessageBrokerRegistry;  import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;  import org.springframework.web.socket.config.annotation.StompEndpointRegistry;  import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;  @Configuration  @EnableWebSocketMessageBroker  public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {  @Override  public void registerStompEndpoints(StompEndpointRegistry registry) {  registry.addEndpoint("/ws").setAllowedOrigins("\*").withSockJS(); // Punto final del WebSocket  }  @Override  public void configureMessageBroker(MessageBrokerRegistry registry) {  registry.enableSimpleBroker("/topic"); // Prefijo para los mensajes enviados al cliente  registry.setApplicationDestinationPrefixes("/app"); // Prefijo para mensajes enviados desde el cliente al servidor  }  } |
| --- |

1. Cree una nueva clase que se llame WebSocketNotificationService en el paquete co.edu.uniquindio.proyecto.servicios.impl así:

| package co.edu.uniquindio.proyecto.servicios.impl;  import co.edu.uniquindio.proyecto.dto.NotificacionDTO;  import lombok.RequiredArgsConstructor;  import org.springframework.messaging.simp.SimpMessagingTemplate;  import org.springframework.stereotype.Service;  @Service  @RequiredArgsConstructor  public class WebSocketNotificationService {  private final SimpMessagingTemplate messagingTemplate;  public void notificarClientes(NotificacionDTO notificacionDTO) {  // Envía un mensaje a todos los clientes suscritos al tema "/topic/reports"  messagingTemplate.convertAndSend("/topic/reports", notificacionDTO);  }  public void notificarClienteEspecifico(String userId, NotificacionDTO notificacion) {  messagingTemplate.convertAndSend("/topic/user/" + userId + "/reports", notificacion);  }  } |
| --- |

Debe crear el DTO NotificacionDTO con los siguientes atributos: título, cuerpo y topic, todos de tipo String.

1. Cuando lo necesite, puede usar el servicio anterior invocando el método notificarClientes(). Por ejemplo, cuando se crea un nuevo reporte puede hacer lo siguiente:

| NotificacionDTO notificacionDTO = new NotificacionDTO(  "Nuevo Reporte",  "Se acaba de crear un nuevo reporte: " + reporte.getTitulo(),  "reports"  );  webSocketNotificationService.notificarClientes(notificacionDTO); |
| --- |

Tenga en cuenta que es necesario que el servicio que haga esto “inyecte” el servicio de WebSocketNotificationService, así como se hace con los repositorios o mappers.

1. Para probar el funcionamiento del WebSocket necesitamos de un archivo de JavaScript que se suscriba al socket y reciba las notificaciones. Este script debe usar STOMP y SockJS Investigue cómo hacerlo.
2. Programe todos los métodos que representen los requerimientos de negocio del proyecto final del espacio académico. Recuerde hacer un manejo adecuado de excepciones y de los DTOs, así mismo haga test a cada método.
3. Recuerde hacer commit de sus avances en el repositorio de git.

| **Para la próxima clase** |
| --- |

* Investigar sobre Spring Security y JWT (JSON Web Token).