Índice de contenido

SPRING BOOT

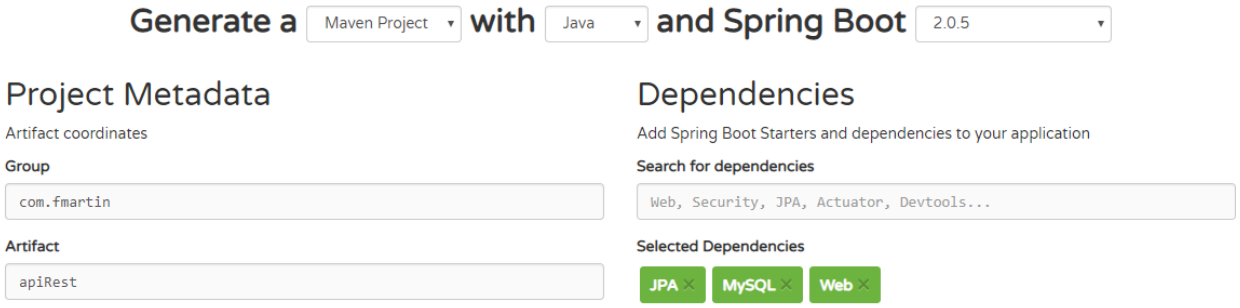
1. CONFIGURACIÓN

1.1. Creación del proyecto:

1.1.1. Creación del proyecto mediante la web de spring boot 'https://start.spring.io/'.

1.1.2. Creación del proyecto mediante ecripse: new/proyect/New Spring Starter Proyect

1.2. Configuración



1.2.1. Group: Ruta inicial de las clases del proyecto.

1.2.2. Artifact: Nombre del fichero de compilación del proyecto.

1.2.3. Dependencies: Podemos introducir las dependencias que queremos introducir en el proyecto.

- Fichero pom.xml

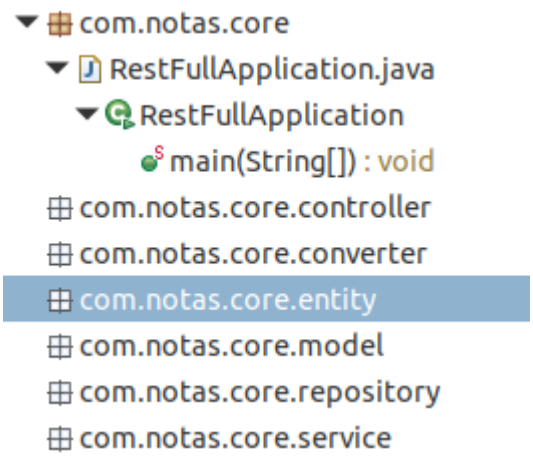
```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <jackson.version>2.8.10</jackson.version>
    <jaxb-api.version>2.2.11</jaxb-api.version>
</properties>

<dependencies>

    <!-- Driver de conexión mysql -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.11</version>
    </dependency>
    <!-- Convierte una clase a Json y Json a una clase -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <!-- Convierte una clase a Xml y Xml a una clase -->
    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>${jaxb-api.version}</version>
        <scope>runtime</scope>
    </dependency>
```

- Fichero de configuración del proyecto 'properties'.

SPRING BOOT

```
#Puerto donde corre el servidor
server.port=8080
#Muestra el error 404
server.error.whitelabel.enabled=false

#Datos de conexión con la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/restSpring?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&useSSL=false
spring.datasource.username=root
spring.datasource.password=Icaro1979++

#Muestra el código sql por la consola
spring.jpa.show-sql=true
#Cuando se corre el proyecto se actualiza/crea/ la Base de Datos con los cambios
spring.jpa.hibernate.ddl-auto=update
#Strategia de la base de datos
spring.jpa.hibernate.naming.strategy=org.hibernate.cfg.ImprovedNamingStrategy
#Indica que tipo de base de datos vamos ha usar.
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

- Estructura de ficheros.



  - Controller:
  - Converter:
  - Entity:
  - Model:
  - Repository:
  - Service:

2. ENTIDADES Y MODELOS.

- Entity.

```java
import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * @author fmgar
 *
 */
@Entity
@Table(name="NOTA")
public class Nota implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 1165474281158866341L;

    @Id
    @GeneratedValue
    @Column(name="ID")
    private Long id;

    @Column(name="NOMBRE", unique=true)
    private String nombre;

    @Column(name="TITULO")
    private String titulo;

    @Column(name="DESCRIPCION")
    private String contenido;

    public Nota() {

    }

    public Nota(Long id, String nombre, String titulo, String contenido) {
        this.id = id;
        this.nombre = nombre;
        this.titulo = titulo;
        this.contenido = contenido;
    }
```

- Permite mapear la Clase con una tabla de la base de datos por medio de anotaciones de JPA.
  - `@Entity`: Definimos que la clase es una entidad.
  - `@Table`: Definimos que la clase mapea una tabla, identificamos la tabla que mapea la clase por medio del nombre.
  - `@Id`: Define que la propiedad anotada representa al campo identificador de la tabla.
  - `@GeneratedValue`: Define como se va ha generar el valor de la propiedad identificadora de la tabla

- `@Column`: Define que la propiedad anotada representa un campo de la tabla, pudiendo definir el nombre de la tabla, el tipo de dato que va a guardar, si es un campo que no se van a repetir los valores (`unique`), etc.

- Si utilizamos la anotación @XmlRootElement antes de la definición de la clase indicamos que vamos a utilizar xml en vez e json para convertir los datos a la clase entidad.
- Se va a trabajar con el por medio de los repositorios.

- Model.

```
import com.notas.core.entity.Nota;

public class MNota {

    private Long id;

    private String nombre;

    private String titulo;

    private String contenido;

    public MNota() {

    }

    public MNota(Long id, String nombre, String titulo, String contenido) {
        this.id = id;
        this.nombre = nombre;
        this.titulo = titulo;
        this.contenido = contenido;
    }

    public MNota(Nota nota) {
        this.id = nota.getId();
        this.nombre = nota.getNombre();
        this.titulo = nota.getTitulo();
        this.contenido = nota.getContenido();
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

- Representa la entidad como el modelo de dato que vamos a devolver al usuario en la vista.
- Debamos crear un constructor que construya el modelo a partir de un objeto de la entidad que representa.
- Se va a trabajar con el por medio de los controladores.

3. REPOSITORIOS Y CONVERTIDORES.

- Repositorio.

```java
package com.notas.core.repository;

import java.io.Serializable;
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.notas.core.entity.Nota;

@Repository("repositorio")
public interface NotaRepository extends JpaRepository<Nota, Serializable>{

    public abstract Nota findByNombre(String nombre);

    public abstract List<Nota> findByTitulo(String titulo);

    public abstract Nota findByNombreAndTitulo(String nombre, String titulo);
}
```

  - Se encarga de hacer peticiones a la base de datos y actualizar los datos mediante entidades relacionadas con las tablas a consultar o actualizar.
  - Se crea mediante una interfaz que extiende de `JpaRepository`, debemos definir la entidad que vamos a utilizar.
  - `@Repository`: Mediante esta anotación definimos que la interfaz es un repositorio y el nombre del bean para inyectar el repositorio en el servicio.
  - Podemos definir métodos abstractos que van a ser los encargados de realizar las consultas a la base de datos y devolvernos una entidad o una lista de entidades rellenas con el resultado de la consulta.
    - NombreEntidad findByNombreCampo: Devuelve una entidad que el valor del campo en la tabla que tiene el nombre del método coincida con el parámetro que le pasamos al método.
    - List<NombreEntidad> findByNombreCampo: Devuelve una lista de entidades en las que el valor del campo en la tabla que tiene el nombre del método coincida con el parámetro que le pasamos al método.
    - NombreEntidad findByNombreCampo1AndNombreCampo2: Devuelve una entidad realizando una consulta en la que coinciden los valores de los parámetros en los campos definidos en el nombre del método.

- Convertidor

```java
package com.notas.core.converter;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Component;

import com.notas.core.entity.Nota;
import com.notas.core.model.MNota;

@Component("convertidor")
public class Convertidor {
    public List<MNota> convertirLista(List<Nota> notas){
        List<MNota> mnotas = new ArrayList<MNota>();
        for(Nota nota: notas) {
            mnotas.add(new MNota(nota));
        }
        return mnotas;
    }

}
```

- @Componet: Mediante esta anotación definimos que la clase es un componente y el nombre del bean para inyectar el componente en el servicio.
- Por medio del método convertirLista convierte una lista de tipo de entidad a un tipo de modelo para devolvérselo al Usuario en la vista.

4. SERVICIO

```java
package com.notas.core.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import com.notas.core.converter.Convertidor;
import com.notas.core.entity.Nota;
import com.notas.core.model.MNota;
import com.notas.core.repository.NotaRepository;

@Service("servicio")
public class NotaService {
    @Autowired
    @Qualifier("repositorio")
    private NotaRepository repositorio;

    @Autowired
    @Qualifier("convertidor")
    private Convertidor convertidor;

    public boolean actualizar(Nota nota) {
        try {
            repositorio.save(nota);
            return true;
        } catch (Exception ex) {
            return false;
        }
    }

    public boolean borrar(String nombre, Long id) {
        try {
            Nota nota = repositorio.findByNombreAndId(nombre, id);
            repositorio.delete(nota);
            return true;
        } catch (Exception ex) {
            return false;
        }
    }

    public List<MNota> obtener() {
        return convertidor.convertirLista(repositorio.findAll());
    }

    public MNota obtenerPorNombreYTitulo(String nombre, String titulo) {
        return new MNota(repositorio.findByNombreAndTitulo(nombre, titulo));
    }
```

- El Servicio define una serie de métodos para interactuar con los repositorios de la aplicación, pudiendo consultar y actualizar datos en las tablas de la base de datos, etc.
- @Service: La anotación indica que la clase es un servicio y define el nombre del bean para inyectar el servicio en el controlador.
- @Autowired: La anotación indicamos a spring que vamos a inyectar un bean
- @Qualifier: Con la anotación indicamos el nombre del bean a inyectar.

5. CONTROLADOR

5.1. Creación de controlador.

```java
package com.notas.core.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.notas.core.service.NotaService;

@RestController
@RequestMapping("/v1")
public class NotaController {
    @Autowired
    @Qualifier("servicio")
    private NotaService servicio;
}
```

- @RestController: Esta anotación indica que esta clase en la controladora de una Api Rest.
- @RequestMapping: Esta anotación mapea el controlador, también indicamos la versión del Api Rest.
- @Autowired y @Qualifier: Por medio de estas anotaciones inyectamos las dependencias que necesitemos de los servicios.

5.2. Método crear.

```java
@PostMapping("/nuevo")
public ResponseEntity<?> create(@RequestBody Producto producto) {
    if (StringUtils.isBlank(producto.getNombre()))
        return new ResponseEntity<Mensaje>(new Mensaje("El nombre es obligatorio"), HttpStatus.BAD_REQUEST);
    if ((Integer) producto.getPrecio() == null || producto.getPrecio() == 0)
        return new ResponseEntity<Mensaje>(new Mensaje("el precio es obligatorio"), HttpStatus.BAD_REQUEST);
    if (productoService.existePorNombre(producto.getNombre()))
        return new ResponseEntity<Mensaje>(new Mensaje("ese nombre ya existe"), HttpStatus.BAD_REQUEST);
    producto = productoService.guardar(producto);
    return new ResponseEntity<Producto>(producto, HttpStatus.CREATED);
}
```

- @PutMapping: Mediante esta anotación indicamos que el método atenderá las peticiones put que nos lleguen por medio de la url /nuevo.
- @RequestBody: Esta anotación nos permite capturar los datos contenidos en el cuerpo de la petición.
- @Valid: Esta anotación permite convertir en json que nos llega en el cuerpo de la petición a una clase que indicamos como parámetro del método.

5.3. Método actualizar.

```
@PutMapping("/actualizar")
public ResponseEntity<?> update(@RequestBody Producto producto) {
    if (!productoService.existePorId(producto.getId()))
        return new ResponseEntity<Mensaje>(new Mensaje("no existe ese producto"), HttpStatus.NOT_FOUND);
    if (StringUtils.isBlank(producto.getNombre()))
        return new ResponseEntity<Mensaje>(new Mensaje("el nombre es obligatorio"), HttpStatus.BAD_REQUEST);
    if ((Integer) producto.getPrecio() == null || producto.getPrecio() == 0)
        return new ResponseEntity<Mensaje>(new Mensaje("el precio es obligatorio"), HttpStatus.BAD_REQUEST);
    if (productoService.existePorNombre(producto.getNombre())
            && productoService.obtenerPorNombre(producto.getNombre()).get().getId() != producto.getId())
        return new ResponseEntity<Mensaje>(new Mensaje("ese nombre ya existe"), HttpStatus.BAD_REQUEST);
    producto = productoService.guardar(producto);
    return new ResponseEntity<Producto>(producto, HttpStatus.CREATED);
}
```

- @PostMapping: Mediante esta anotación indicamos que el método atenderá las peticiones post que nos lleguen por medio de la url /actualizar.
- @RequestBody: Esta anotación nos permite capturar los datos contenidos en el cuerpo de la petición.
- @Valid: Esta anotación permite convertir en json que nos llega en el cuerpo de la petición a una clase que indicamos como parámetro del método.

5.4. Método eliminar.

```
@DeleteMapping("/borrar/{id}")
public ResponseEntity<?> delete(@PathVariable Long id) {
    if (!productoService.existePorId(id))
        return new ResponseEntity<Mensaje>(new Mensaje("no existe ese producto"), HttpStatus.NOT_FOUND);
    productoService.borrar(id);
    return new ResponseEntity<Mensaje>(new Mensaje("producto eliminado"), HttpStatus.OK);
}
```

- @DeleteMapping: Mediante esta anotación indicamos que el método atenderá las peticiones delete que nos lleguen por medio de la url /borrar. En la url le indicamos los parámetros que va a recibir en la petición entre /{ nombreParametro}
- @PathVariable: Mediante esta anotación recuperamos un valor de un parámetro indicando el nombre del parámetro a recuperar.

5.5. Método consultar una lista

```
@GetMapping("/notas")
public List<MNota> obtenerNotas(){
    return servicio.obtener();
}
```

- @GetMapping: Mediante esta anotación indicamos que el método atenderá las peticiones get que nos lleguen por medio de la url /notas y devolverá una lista del modelo de datos.

5.6. Método consulta de una entidad

```
@GetMapping("/detalle/{id}")
public ResponseEntity<?> getOne(@PathVariable Long id) {
    if (!productoService.existePorId(id)) {
        return new ResponseEntity<Mensaje>(new Mensaje("No existe ese producto"), HttpStatus.NOT_FOUND);
    } else {
        Producto producto = productoService.obtenerPorId(id).get();
        return new ResponseEntity<Producto>(producto, HttpStatus.OK);
    }
}
```

- @GetMapping: Mediante esta anotación indicamos que el método atenderá las peticiones get que nos lleguen por medio de la url /detalle y devolverá una entidad por su id
- @PathVariable: Mediante esta anotación recuperamos un valor de un parámetro indicando el nombre del parámetro a recuperar.

5.7. HttpStatus.

NOT_FOUND	No existe la entidad que solicitamos en el sistema mediante la petición.
BAD_REQUEST	Hay un error de validación de la petición que hacemos al sistema.
OK	La petición se ha realizado correctamente.
CREATED	La petición de crear / modificar entidad se ha realizado correctamente.

5.8. CrossOrigin.

- Para habilitar la política cors en el controller del REST. Al inicio de la clase Controller ponemos la anotación de @CrossOrigin y la ruta origen con el puerto.

```
@CrossOrigin(origins = "[Ruta_Servidor]")
```

Ejemplo: `@CrossOrigin(origins = "http://localhost:4200")`

6. GESTIÓN DE LOGS

```java
private static final Log logger = LogFactory.getLog(NotaService.class);

public boolean actualizar(Nota nota) {
    try {
        logger.info("ACTUALIZANDO NOTA");
        repositorio.save(nota);
        logger.info("NOTA ACTUALIZADA");
        return true;
    } catch (Exception ex) {
        logger.error("HUBO UN ERROR");
        return false;
    }
}
```

- Por medio de la clase Log podemos escribir en el log de la aplicación, para inicializar la clase utilizamos LogFactory.getLog(Nobre_Clase.class)
- Tipos de logs:
  - info: Muestra información en el log.
  - error: Muestra un error en el log.
  - warn: Muestra una alerta en el log.

7. PAGINACIÓN

- Configuración en fichero de propiedades '.properties'.

```
#Paginación
spring.data.rest.page-param-name=page
spring.data.rest.limit-param-name=limit
#spring.data.rest.sort-param-name=sort
spring.data.rest.default-page-size=5
spring.data.rest.max-page-size=10
```

  - Propiedad page-param-name: Indica el nombre del parámetro que se va a utilizar para indicar en que página estamos actualmente.
  - Propiedad limit-param-name: Indica el nombre del parámetro que define el límite de valores que se van a devolver en la consulta.
  - Propiedad sort-param-name:
  - Propiedad default-page-size: Indica el tamaño por defecto de la paginación.
  - Propiedad max-page-size: Indica la cantidad máxima de registros que se van a enviar.

- Repositorio.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import com.notas.core.entity.Nota;

@Repository("repositorio")
public interface NotaRepository extends JpaRepository<Nota, Serializable>, PagingAndSortingRepository<Nota, Serializable>{
    public abstract Page<Nota> findAll(Pageable pageable);
```

  - La interfaz debe extender también a PagingAndSortingRepository, hay que indicar la entidad con al que va a trabajar.
  - El método abstracto de devolución de la lista devuelve un objeto de tipo Page en donde le indicamos la entidad con la que va a trabajar y recibe como parámetro un objeto de tipo Pageable que contiene toda la información referente a la paginación.

- Servicio.

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import com.notas.core.converter.Convertidor;
import com.notas.core.entity.Nota;
import com.notas.core.model.MNota;
import com.notas.core.repository.NotaRepository;

@Service("servicio")
public class NotaService {
    @Autowired
    @Qualifier("repositorio")
    private NotaRepository repositorio;

    @Autowired
    @Qualifier("convertidor")
    private Convertidor convertidor;

    public List<MNota> obtenerPorPaginacion(Pageable pageable){
        return convertidor.convertirLista(repositorio.findAll(pageable).getContent());

    }
```

- El método que devuelve la lista recibe un parámetro de tipo Pageable y retorna la lista que devuelve el método del repositorio pasando como parámetro el parámetro de tipo Pageable que ha recibido.
- El método del servicio devuelve un objeto de tipo Page y llamando al método getContent() se puede recuperar la lista.

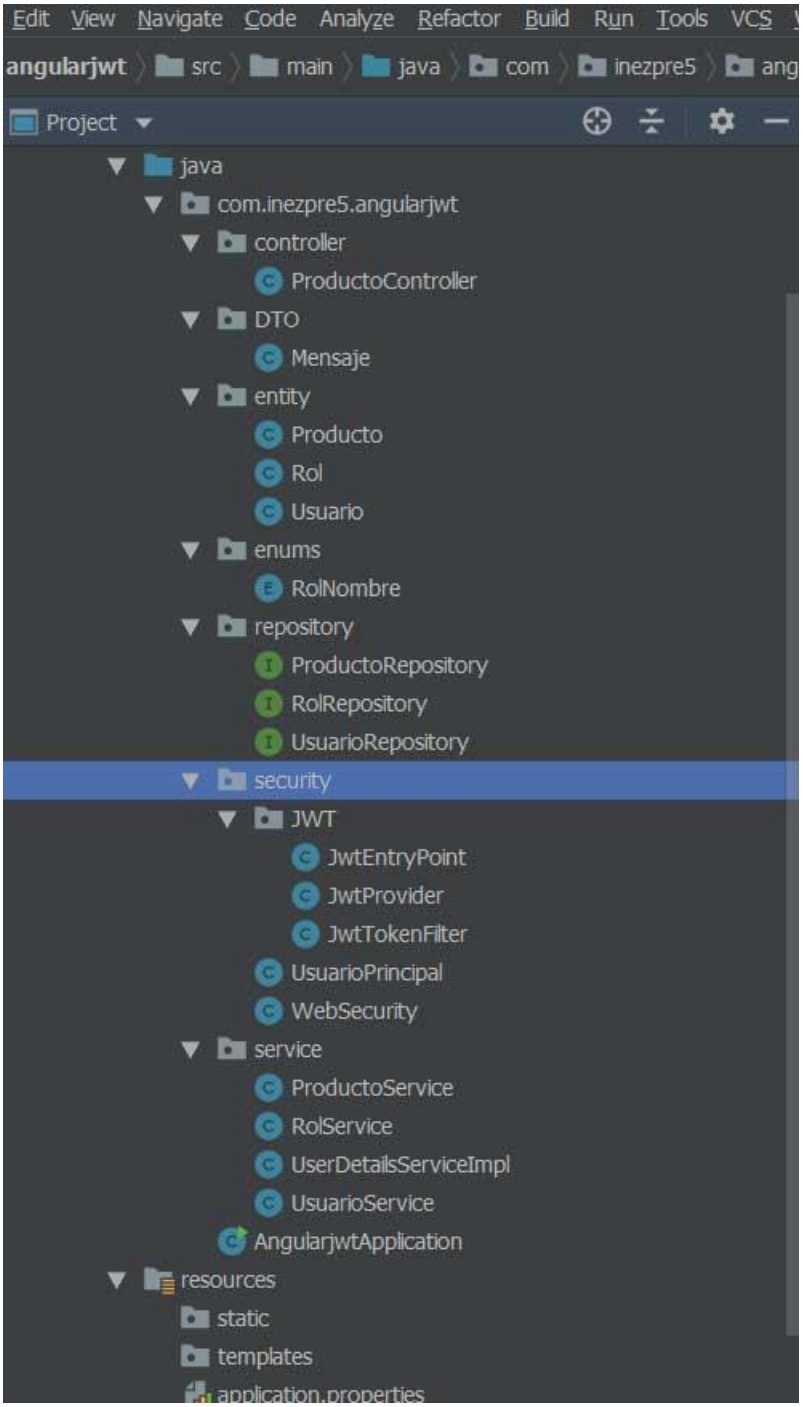
- Controlador

```java
import com.notas.core.entity.Nota;
import com.notas.core.service.NotaService;
import com.notas.core.model.MNota;

@RestController
@RequestMapping("/v1")
public class NotaController {
    @Autowired
    @Qualifier("servicio")
    private NotaService servicio;

    @GetMapping("/notas-paginacion")
    public List<MNota> obtenerNotasPaginadas(Pageable pageable){
        return servicio.obtenerPorPaginacion(pageable);
    }
```

- El método que mapea la dirección '/notas-paginacion' recibe como parámetro un objeto de tipo Pageable que después pasa como parámetro al método del servicio que devuelve la lista.
- Ejemplo de url de petición http://localhost:8080/v1/notas-paginacion?page=1&size=8.
  - El parámetro page le indica la página actual donde estamos.

- El parámetro `size` le indica el número máximo de registros por página que vamos a recibir del controlador.
- Si no le pasamos los parámetros devuelve todos los registros de la lista.

8. SEGURIDAD JWT TOKEN.



8.1. Usuario y Rol

- Enum con los diferentes tipos de roles de la aplicación

```
package com.inezpre5.angularjwt.enums;


public enum RolNombre {

    ROLE_ADMIN,

    ROLE_USER

}
```

- Clase Rol guarda los diferentes tipos que rol que existen en la aplicación

```
package com.inezpre5.angularjwt.entity;

import com.inezpre5.angularjwt.enums.RolNombre;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
public class Rol {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @NotNull
    private RolNombre rolNombre;

    public Rol() {
    }

    public Rol(@NotNull RolNombre rolNombre) {
        this.rolNombre = rolNombre;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public RolNombre getRolNombre() {
        return rolNombre;
    }

    public void setRolNombre(RolNombre rolNombre) {
```

```
        this.rolNombre = rolNombre;
    }
}
```

- La anotación @Enumerated indica que es un campo que se corresponde con un enum tipo String, RolNombre

- La Clase usuario guarda los datos referente a un usuario,

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    private String nombre;

    @NotNull
    @Column(unique = true)
    private String nombreUsuario;

    @NotNull
    @Column(unique = true)
    private String email;

    @NotNull
    private String password;

    @NotNull
    @ManyToMany
    @JoinTable(name = "usuario_rol", joinColumns = @JoinColumn(name = "usuario_id"),
inverseJoinColumns = @JoinColumn(name = "rol_id"))
    private Set<Rol> roles = new HashSet<>();

    public Usuario() {
    }

    public Usuario(@NotNull String nombre, @NotNull String nombreUsuario, @NotNull
String email, @NotNull String password) {
        this.nombre = nombre;
        this.nombreUsuario = nombreUsuario;
        this.email = email;
        this.password = password;
    }

/*GET Y SET*/
```

- En el constructor no se debe incluir los campos id y roles.

- Usuario Repository contiene métodos para comprobar el nombre del usuario y el email son únicos.2

@Repository

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
   Optional<Usuario> findByNombreUsuario(String nu);
   boolean existsByNombreUsuario(String nu);
   boolean existsByEmail(String email);
}
```

- Rol Repository

```
@Repository
public interface RolRepository extends JpaRepository<Rol, Long> {
    Optional<Rol> findByRolNombre(RolNombre rolNombre);
}
```

- Servicio Usuario implementa lo métodos de comprobación de si existe un usuario con un nombre o con un email específico, además de guardar y obtener un Opcional por nombre de usuario.

```
@Service
@Transactional
public class UsuarioService {

    @Autowired
    UsuarioRepository usuarioRepository;

    public Optional<Usuario> getByNombreUsuario(String nu){
        return usuarioRepository.findByNombreUsuario(nu);
    }

    public boolean existePorNombre(String nu){
        return usuarioRepository.existsByNombreUsuario(nu);
    }

    public  boolean existePorEmail(String email){
        return usuarioRepository.existsByEmail(email);
    }

    public void guardar(Usuario usuario){
        usuarioRepository.save(usuario);
```

```
        }
    }
```

- Servicio Rol implementa un método, para obtener un Rol a partir de un objeto tipo RolNombre.

```
Service
@Transactional
public class RolService {

    @Autowired
    RolRepository rolRepository;

    public Optional<Rol> getByRolNombre(RolNombre rolNombre){
        return rolRepository.findByRolNombre(rolNombre);
    }
}
```

8.2. Class de Spring Security y JWT (UserDetails y UserDetailsServiceImpl)

- Dependencias Maven
    - Spring Starter Security
    - Jsonwebtoken

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
            </dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt -->
            <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt</artifactId>
                <version>0.9.0</version>
            </dependency>
```

- Usuario principal implementa la interfaz UserDetails.

```
import org.springframework.security.core.GrantedAuthority;
```

```
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class UsuarioPrincipal implements UserDetails {

    private Long id;
    private String nombre;
    private String nombreUsuario;
    private String email;
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    public UsuarioPrincipal(Long id, String nombre, String nombreUsuario,
        String email, String password,
        Collection<? extends  GrantedAuthority> authorities) {
        this.id = id;
        this.nombre = nombre;
        this.nombreUsuario = nombreUsuario;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }

    public static UsuarioPrincipal build(Usuario usuario){
        List<GrantedAuthority> authorities =
                usuario.getRoles().stream().map(
                rol -> new SimpleGrantedAuthority(rol.getRolNombre().name()))
                .collect(Collectors.toList());
        return new UsuarioPrincipal(usuario.getId(), usuario.getNombre(),
           usuario.getNombreUsuario(), usuario.getEmail(),
           usuario.getPassword(), authorities);
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
```

```
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}

/*GET Y SET*/
```

- La clase se utiliza para la autenticación.
- En el método estático build, se crea una lista de privilegios (GrantedAutority) y se le asignan en el constructor al UsuarioPrincipal que devuelve le método.

- UserDetailsServiceImpl implementa la interfaz loadUserDetailsService. El método de la interfaz loadUserByUsername sirve para obtener un usuario a partir de un nombre.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
```

```
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    UsuarioService usuarioService;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(String nombreUsuario)
       throws UsernameNotFoundException {
        Usuario usuario = usuarioService.getByNombreUsuario(nombreUsuario).get();
        return UsuarioPrincipal.build(usuario);
    }
}
```

8.3. Json Web Token.

- En el paquete JWT hay tres clases:
  - JwtEntryPoint
  - JwtProvider
  - JwtTokenFilter.

- JwtEntryPoint. La clase debe implementar la interfaz AuthenticationEntryPoint

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
@Component
public class JwtEntryPoint implements AuthenticationEntryPoint {

    private static  final Logger logger = LoggerFactory.getLogger(
           JwtEntryPoint.class);
    @Override
    public void commence(HttpServletRequest req, HttpServletResponse res,
        AuthenticationException e) throws IOException, ServletException {
        logger.error("fail en el método commence");
        res.sendError(HttpServletResponse.SC_UNAUTHORIZED, "credenciales erróneas");
    }
}
```

- En esta clase se comprueba las credenciales en el login y se imprime por consola el error y comprueba cual es el método que produce el error.

- JwtProvider
  - En el fichero application.properties añadimos dos valores secret y expiration

```
# values
jwt.secret = secret
jwt.expiration = 36000
```

- La clase.

```
import io.jsonwebtoken.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtProvider {
```

```
private static  final Logger logger = LoggerFactory.getLogger(
 JwtEntryPoint.class);

@Value("${jwt.secret}")
private String secret;

@Value("${jwt.expiration}")
private int expiration;

public String generateToken(Authentication authentication) {
    UsuarioPrincipal usuarioPrincipal =
      (UsuarioPrincipal) authentication.getPrincipal();
    return Jwts.builder().setSubject(usuarioPrincipal.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(new Date().getTime() + expiration * 1000))
            .signWith(SignatureAlgorithm.HS512, secret)
            .compact();
}

public String getNombreUsuarioFromToken(String token){
    return Jwts.parser().setSigningKey(secret).parseClaimsJws(token)
          .getBody().getSubject();
}

public boolean validateToken(String token) {
    try {
        Jwts.parser().setSigningKey(secret).parseClaimsJws(token);
        return true;
    } catch (MalformedJwtException e) {
        logger.error("token mal formado " +e.getMessage());
    } catch (UnsupportedJwtException e) {
        logger.error("token no soportado " +e.getMessage());
    } catch (ExpiredJwtException e) {
        logger.error("token expirado " +e.getMessage());
```

```
        } catch (IllegalArgumentException e) {
            logger.error("token vacío " +e.getMessage());
        } catch (SignatureException e) {
            logger.error("error en la firma " +e.getMessage());
        }
        return false;
    }
}
```

- Esta clase controla la creación del Token, validación del token y retorno del nombre de usuario a partir del token. En la generación:
  - Se asigna el nombre del usuario (setSubject()).
  - La fecha de creación.
  - Expiración y se firma (setSignInkey()).

- JwtTokenFilter extiende de la clase OncePerRequestFilter.

```
package com.inezpre5.angularjwt.security.JWT;

import com.inezpre5.angularjwt.service.UserDetailsServiceImpl;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class JwtTokenFilter extends OncePerRequestFilter {
```

```
private static  final Logger logger =
   LoggerFactory.getLogger(JwtTokenFilter.class);

@Autowired
JwtProvider jwtProvider;

@Autowired
UserDetailsServiceImpl userDetailsServiceImpl;

@Override
protected void doFilterInternal(HttpServletRequest req,
   HttpServletResponse res, FilterChain filterChain)
   throws ServletException, IOException {
    try {
        String token = getToken(req);
        if(token !=null && jwtProvider.validateToken(token)){
            String nombreUsuario = jwtProvider.getNombreUsuarioFromToken(token);
            UserDetails userDetails =
               userDetailsServiceImpl.loadUserByUsername(nombreUsuario);
            UsernamePasswordAuthenticationToken auth =
               new UsernamePasswordAuthenticationToken(userDetails, null,
               userDetails.getAuthorities());
            SecurityContextHolder.getContext().setAuthentication(auth);
        }
    }catch (Exception e){
        logger.error("fail en método doFilter " + e.getMessage());
    }
    filterChain.doFilter(req, res);
}

private String getToken(HttpServletRequest request){
    String authReq = request.getHeader("Authorization");
    if(authReq != null && authReq.startsWith("Bearer "))
        return authReq.replace("Bearer ", "");
    return null;
```

```
    }
}
```

- En esta clase se obtiene el token y el usuario por medio del nombre de usuario (Codigo) y comprueba si es la autenticación es correcta. En caso afirmativo la añade al contexto y se añade al filtro.
- Con getToken() se elimina el inicio “Bearer ”.

8.4. WebSecurity

9. TEST JUNIT

```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.notas.core.model.MNota;
import com.notas.core.service.NotaService;

@RunWith(SpringRunner.class)
@SpringBootTest
public class RestFullApplicationTests {
    @Autowired
    @Qualifier("servicio")
    private NotaService servicio;

    @Test
    public void contextLoads() {
        List<MNota> lista = servicio.obtener();
    }

}
```

- Debemos anotar la clase de test con las anotaciones:
  - @RunWith(SpringRunner.class) que indica que es una clase ejecutable de Spring.
  - @SpringBootTest que indica que es una clase de test de Spring Boot