**SpringBoot**

Es una herramienta que permite el inicio rápido del desarrollo de aplicaciones basadas en Spring haciendo muy fácil la inclusión de librerías de terceros.

**Starters.** Se utilizan para resolver el difícil problema de la gestión de dependencias de librerías de terceros**. Gestionan las dependencias de los proyectos incluidos en el nuestro haciendo que si son diferentes puedan encajar.**

* Validation. Usado para validar los dtos con Hibernate Validator.
* Spring Data JPA. Spring Data JPA con Hibernate.
* Spring Web. Usa Spring MVC, REST y Tomcat como servidor integrado predeterminado.
* Spring Security
* H2 Database. Como BD de test.
* MS SQL Server

**Anotaciones**

@SpringBootApplication

En Spring Boot, es muy común que se apliquen múltiples anotaciones sobre la clase principal (main). Pero comúnmente, la mayoría de los desarrolladores aplican principalmente 3 anotaciones.

* **@Configuration:** Disponible a partir de la versión 3 de Spring, nos ofrece la posibilidad de realizar una notación que será la encargada de definir a la clase que lo posea como una clase de configuración. Esta configuración para el framework de Spring estará basada en anotaciones. Y no como en sus orígenes, que estaba basada en XML, lo que lo hacía más complejo. La finalidad de dicha anotación también será el permitir realizar la inyección de dependencias.
* **@EnableAutoConfiguration:** La configuración automática de Spring Boot, intenta configurar automáticamente su aplicación Spring en función de las dependencias jar que haya agregado. Si por ejemplo, si HSQLDB (sistema gestor de bases de datos) está en su ruta de clase y no ha configurado manualmente ningún bean de conexión de base de datos, Spring Boot configura automáticamente una base de datos en memoria.
* **@ComponentScan:** Se utiliza junto a @Configuration para indicar a Spring donde debe buscar los componentes y será dentro del package que tenemos anotado. Por solo tener que anotarla una vez, poder hacer que todos los packages sean hijos del package de la clase padre (el que contenga el main).

La gran mayoría de desarrolladores de Spring Boot, suelen tener en su clase principal (main) anotada con las anotaciones @Configuration, @EnableAutoConfiguration y @ComponentScan. Dado que estas anotaciones se combinan con tanta frecuencia, Spring Boot ofrece el substituir tener que declarar constantemente este tridente de anotaciones, por declarar únicamente una sola **@SpringBootApplication** que englobará a las 3 anteriores.

@RestController

Es esencialmente una combinación de @Controller y @ResponseBody. La función principal es permitir que una clase sea reconocida como un componente administrado por Spring y permitir el manejo de solicitudes HTTP usando la API REST.

@RequestMapping

Se utiliza para asignar solicitudes web a los métodos de Spring Controller.

Parámetros que puede usar:

value: para indicar la url que procesará. Ej. value = “/url/{param}” Se puede indicar una url o varias, aunque lo normal es incluir una. También se pueden indicar como valor un asterisco, que hará que se procesen todas las peticiones.

Como podemos ver se pueden incluir parámetros (únicos o múltiples). Para hacer esto luego se debe incluir la anotación @PathVariable en la variable del método que procesará esta petición. Se puede incluir tras la etiqueta el nombre de la variable (“id”) o no incluirlo si coincide con el nombre del parámetro del método. También se pueden incluir las variables como parámetro de la anotación @RequestMapping usando el atributo “params” y da la posibilidad de incluir varios y no es necesario usar todos.

También se pueden usar expresiones regulares. Ej. value = "/ex/bars/{numericId:[\\d]+}" (esto hace que el parámetro numericId deba ser numérico)

Hay que tener cuidado para no se duplique el valor del parámetro value en distintos métodos. Esto produciría una excepción. Se produciría si tiene el mismo valor de value, method, parámetros, headers y media type.

method: get, post, update, delete, … Si no se indica ninguno podrá procesar todos y se pueden indicar varios.

headers: se pueden reducir aún mas las asignaciones especificando parámetros en la cabecera. Se puede incluir uno o varios.

produces: Indica el tipo de resultado que produce, (application-json, application.xml)

@Autowired

Permite que Spring resuelva e inyecte beans de colaboración en nuestro bean. Se puede usar en propiedades, setter y constructores. Si se usa en propiedades no son necesarios los get y los set.

Se puede producir una excepción si inyectamos un bean que está implementado por dos clases distintas. En este caso habría que usar la anotación @Qualifier para indicar a cuál nos referimos exactamente. También podemos crear nuestra propia etiqueta Qualifier.

También se puede usar el nombre del atributo como identificador de la clase que será la implementación. Por ejemplo, si la clase que implementa la clase Padre se llama Abc, el atributo se debería llamar private Padre abc.

@PostMapping, @GetMapping, @PutMapping, @DeleteMapping, @PatchMapping

Se usan para sustituir la anotación @RequestMapping. Ej. @GetMapping("/get/{id}"). Como se puede ver por el nombre de cada anotación, cada una de ellas se usa para recibir un tipo de petición.

@RequestBody

Usando esta anotación en un parámetro de un método, Spring, automáticamente asigna el body de una petición http a un objeto java, asumiendo que la se deserializa el objeto entrante en uno apropiado existente.

@ResponseBody

Con esta anotación se indica al controlador que el objeto resultante de la ejecución del método sea automáticamente serializado en un JSON y pasado en la respuesta. Usando la anotación podemos explicitar el tipo de contenido que devuelve el método. Para esto usamos el atributo produces de las anotaciones GetMapping, PostMapping, etc.

Cuando usamos @RestController en lugar de @Controller no es necesario usar esta anotación ya que Spring lo hace por defecto.

@RequestParam

Podemos usar @RequestParam para extraer parámetros de consulta, parámetros de formulario e incluso archivos de la solicitud.

Si el parámetro pasado se llama igual que en el método que lo recibe se puede poner entre paréntesis el nombre (“nombre”) o no poner nada. Si no se llama igual se debe poner (name = “nombre”)

Los parámetros indicados en la anotación son obligatorios, si no son enviados obtendríamos un error. En caso de hacer que sean opcionales se debe indicar: @RequestParam(required = false). En ese caso el parámetro se cargará con el valor Null.

A partir de java 8 no es necesario usar el parámetro required, se puede usar Optional<class>. En este caso, si no viene cargado el parámetro, se usará el valor por defecto.

También se puede usar un valor por defecto usando @RequestParam(defaultValue = “valor por defecto”). Cuando se usa esta opción se configura por defecto el valor de required a false.

Podemos usar múltiples parámetros usando un Map: @RequestParam Map<String,String> allParams (Parameters are {[name=abc], [id=123]})

O también se puede usar una lista: @RequestParam List<String > id. Ids son [1,2,3]

@PreAuthorize, @PostAuthorize

La anotación @PreAuthorize verifica la expresión dada antes de ingresar al método, mientras que la anotación @PostAuthorize la verifica después de la ejecución del método y podría alterar el resultado.

En ambas también se puede usar para comprobar el nombre el usuario: PreAuthorize("#username == authentication.principal.username")

@PostAuthorize también puede acceder al resultado de la ejecución del método: @PostAuthorize("returnObject.username == authentication.principal.nickName")

@RolesAllowed

Se incluyen como parámetros el rol o roles que pueden invocar el método.

@Entity

Se utilizan para representar objetos java que se corresponden con los datos de una tabla de BD. La entidad debe tener un constructor sin argumentos y una PK. No deben ser declaradas finales. El nombre por defecto de la entidad es el nombre de la clase; Podemos cambiarlo con el parámetro “name” de la anotación.

@Table

Se usa cuando el nombre de la entidad y de la tabla de la BD no es el mismo.

@Id

Cada entidad debe tener un campo Id para identificarla. Además de definir un campo PK hay que indicar el tipo de estrategia de generación de la PK, que podrá ser AUTO, TABLE, SEQUENCE O IDENTITY usando la anotación @GeneratedValue

@Column

Se usa para indicar con que columna se relaciona el campo del objeto java. Si el nombre del campo y la columna es el mismo no es necesaria.

Los posibles parámetros que puede usar son:

@Column(name="STUDENT\_NAME", length=50, nullable=false, unique=false)

@Transient

Se usa para hacer un campo no persistente.

@NotNull

Se usa para validar que la propiedad anotada no es nula.

@ManyToMany

Se usa para indicar una relación de muchos a muchos entre dos entidades.

Para indicar cual de las dos entidades es el propietario de la relación se usa en la tabla estudiante:

@ManyToMany

@JoinTable(

name = "course\_like",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

Set<Course> likedCourses;

name es el nombre la tabla de relación entre las dos entidades.

joinColumns se usa para indicar el id de la entidad propietaria (en este caso estudiante) en la tabla de relación.

inverseJoinColumns se usa para indicar el id de la otra entidad en la tabla de relación.

Ni JoinTable ni joinColumns no imprescindibles, ya que JPA creará las tablas de relación necesarias pero los nombres podrían no coincidir.

En el otro objeto se mapearía así:

@ManyToMany(mappedBy = "likedCourses")

Set<Student> likes;

@JsonIgnore

Se usa para marcar una propiedad de un objeto que será ignorada al deserializar.

@Size

Se usa para validar que la propiedad anotada tiene un valor entre max y min. Se puede usar en String, Colecciones, Map y Arrays.

@Email

Se usa para validar que la propiedad anotada es un email válido.

@JoinTable

Se usa en un many to many para indicar la tabla que se forma (o formará) como relación entre dos tablas.

@RestControllerAdvice

Se usa para el manejo centralizado de excepciones. Se usa a nivel de clase. Se detecta a través de auto escaneo del classpath.

@ExceptionHandler

Es una anotación a nivel de método en el que se indica que hacer cuando se produce la excepción que maneja en cualquier controlador de la app.

@Component, @Repository, @Service

@Component es un estereotipo genérico para cualquier componente administrado por Spring.

@Service anota clases en la capa de servicio.

@Repository anota clases en la capa de persistencia, que actuará como un repositorio de la base de datos.

@Component se usa para marcar los beans como componentes administrados por Spring. @Service y @Repository son casos especiales de @Component.

El trabajo de @Repository es detectar excepciones específicas de persistencia y volver a lanzarlas como una de las excepciones unificadas no verificadas de Spring.

@ServiceMarcamos beans con @Service para indicar que tienen la lógica empresarial. Además de usarse en la capa de servicio, no hay ningún otro uso especial para esta anotación.

@Configuration

Las clases de configuración pueden contener métodos de definición de beans anotados con @Bean.

@EnableWebSecurity

La anotación @EnableWebSecurity es crucial si deshabilitamos la configuración de seguridad predeterminada.

La aplicación no se iniciará si falta. Entonces, la anotación solo es opcional si simplemente anulamos el comportamiento predeterminado usando un WebSecurityConfigurerAdapter.

Es necesario usar PasswordEncoder si usamos Spring Boot 2.

@EnableGlobalMethodSecurity

Permitir la seguridad a nivel de métodos. (anotándolos)

@Bean

En Spring, los objetos que forman la columna vertebral de su aplicación y que son administrados por el contenedor Spring IoC se denominan beans. Un bean es un objeto que es instanciado, ensamblado y administrado por un contenedor Spring IoC. La inversión de control (IoC) es un proceso en el que un objeto define sus dependencias sin crearlas. Este objeto delega el trabajo de construir tales dependencias a un contenedor de IoC.

@Transactional

La anotación también admite configuraciones adicionales:

* el tipo de propagación de la transacción
* el nivel de aislamiento de la transacción
* un TimeOut para la operación envuelta por la transacción
* un indicador readOnly: una sugerencia para el proveedor de persistencia de que la transacción debe ser de solo lectura
* las reglas de RollBack para la transacción.

En un nivel alto, Spring crea proxis para todas las clases anotadas con @Transactional, ya sea en la clase o en cualquiera de los métodos. El proxy permite que el marco inyecte lógica transaccional antes y después del método de ejecución, principalmente para iniciar y confirmar la transacción.

Lo que es importante tener en cuenta es que, si el bean transaccional está implementando una interfaz, por defecto el proxy será un Java Dynamic Proxy. Esto significa que solo se interceptarán las llamadas a métodos externos que entren a través del proxy. Las llamadas de auto invocación no iniciarán ninguna transacción, incluso si el método tiene la anotación @Transactional.

Solo los métodos públicos harán caso a la anotación, el resto la ignora.

@Modifying

Se usa para las consultas de modificación. La anotación @Modifying se utiliza para mejorar la anotación @Query para ejecutar no solo las consultas SELECT, sino también las consultas INSERT, UPDATE, DELETE e incluso DDL.

@Query

Nos brinda la posibilidad de escribir una consulta JPQL o SQL específica en la anotación @Query. Se usa para select, si tenemos que hacer actualizaciones, creaciones y borrados hay que añadir @Modifying.

Se puede usar el campo value para incluir la Query y el campo native para indicar que la Query está hecha con sql nativo.

@Param

Usamos la anotación @Param en la declaración del método para hacer coincidir los parámetros definidos por el nombre en JPQL con los parámetros de la declaración del método. Si el nombre del parámetro en el método no coincide con el que estamos usando en la Query se usaría lo siguiente:

@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")

User findUserByUserStatusAndUserName(@Param("status") Integer userStatus, @Param("name") String userName);

@SpringBootTest

La anotación @SpringBootTest es útil cuando necesitamos arrancar todo el contenedor. La anotación funciona creando el ApplicationContext que se utilizará en nuestras pruebas. Podemos usar el atributo webEnvironment de @SpringBootTest para configurar nuestro entorno de ejecución; Por ejemplo, podemos usar WebEnvironment.MOCK para que el contenedor funcione en un entorno de servlet simulado.

@AutoConfigureMockMvc

Anotación que se puede aplicar a una clase de prueba para habilitar y configurar la configuración automática de MockMvc.

@TestMethodOrder

@TestMethodOrder es una anotación de nivel de tipo que se utiliza para configurar un MethodOrderer para los métodos de prueba de la clase de prueba anotada o la interfaz de prueba.

En este contexto, el término "método de prueba" se refiere a cualquier método anotado con @Test, @RepeatedTest, @ParameterizedTest, @TestFactory o @TestTemplate.

Si @TestMethodOrder no se declara explícitamente en una clase de prueba, se hereda de una clase principal o se declara en una interfaz de prueba implementada por una clase de prueba, los métodos de prueba se ordenarán utilizando un algoritmo predeterminado que es determinista pero intencionalmente no obvio.

@Test

Se usa para indicar que el método anotado es un método de pruebas.

@Order (en los test)

@Order es una anotación que se utiliza para configurar el orden en el que el elemento anotado (es decir, campo o método) debe evaluarse o ejecutarse en relación con otros elementos de la misma categoría.

Cuando se usa con @RegisterExtension, la categoría se aplica a los campos de extensión. Cuando se utiliza con MethodOrderer.OrderAnnotation MethodOrderer, la categoría se aplica a los métodos de prueba.

Si @Order no se declara explícitamente en un elemento, el valor de pedido DEFAULT se asignará al elemento.

@Override

Indica que una declaración de método está destinada a invalidar una declaración de método en un supertipo. Si un método está anotado con este tipo de anotación, los compiladores deben generar un mensaje de error a menos que se cumpla al menos una de las siguientes condiciones:

* El método anula o implementa un método declarado en un supertipo.
* El método tiene una firma que es equivalente a la invalidación de cualquier método público declarado en Object.

**Versión de Java**

Versión de compilador de java. 11

**Sql Server**

SQL Server 2012

SQL Management 11.0.2100.60

**JWT**

Para construir el token usamos la factoría Jwts.builder() con los siguientes valores:

* (**setSubject**): Establece el valor secundario (sujeto) de reclamaciones de JWT. Estamos usando el username de quién solicita el token.
* (**setIssuer**): Establece el valor del emisor de reclamaciones de JWT (emisor). Usamos como valor del emisor https://www.ezentis.com/
* (**claim**): Establece un valor de parámetro de reclamaciones de JWT personalizado. Se añade al inicio del token el String Authorization.
* (**signWith**): Firma el JWT construido usando el algoritmo especificado con la clave especificada, produciendo un JWS. Se utiliza una función Hash 512 para construir el token.
* (**setIssuedAt**): Establece el valor iat (emitido en) de las reclamaciones de JWT. Es la fecha en la que se crea el token.
* (**setExpiration**). Establece el valor de expiración (vencimiento) de las reclamaciones de JWT. El valor establecido es de 5 horas.
* (**compact**()): Realmente construye el JWT y lo serializa en una cadena compacta y segura para URL de acuerdo con las reglas de serialización compacta de JWT.

BCryptPasswordEncoder

Otras implementaciones requieren que el cliente pase un valor de Salt como argumento,BCrypt, sin embargo, generará internamente una salt aleatoria en su lugar. Es importante entender esto porque significa que cada llamada tendrá un resultado diferente, por lo que solo necesitamos codificar la contraseña una vez.

Para que esta generación de sal aleatoria funcione, BCrypt almacenará la sal dentro del valor hash. Por ejemplo, en el siguiente valor hash:

$2a$10$ZLhnHxdpHETcxmtEStgpI./Ri1mksgJ9iDP36FmfMdYyVg9g0b2dq

Hay tres campos separados por $:

* El "2a" representa la versión del algoritmo BCrypt
* El "10" representa la fuerza del algoritmo
* El "ZLhnHxdpHETcxmtEStgpI". parte es en realidad la sal generada aleatoriamente. Básicamente, los primeros 22 caracteres son sal. La parte restante del último campo es la versión hash real del texto sin formato

Además, tenga en cuenta que el algoritmo BCrypt genera una Cadena de longitud 60, por lo que debemos asegurarnos de que la contraseña se almacenará en una columna que pueda acomodarla.

org.springframework.security.authentication.AuthenticationManager

En pocas palabras, AuthenticationManager es la interfaz que maneja la estrategia principal para la autenticación. Si el principal de la autenticación de entrada es válido y está verificado, AuthenticationManager # authenticate devuelve una instancia de autenticación con el indicador autenticado establecido en verdadero. De lo contrario, si el principal no es válido, lanzará una AuthenticationException. Para el último caso, devuelve nulo si no puede decidir.

ProviderManager es la implementación predeterminada de AuthenticationManager. Delega el proceso de autenticación a una lista de instancias de AuthenticationProvider. Podemos configurar AuthenticationManager global o local si ampliamos WebSecurityConfigurerAdapter.

org.springframework.security.core.context.SecurityContextHolder

Asocia un SecurityContext determinado con el subproceso de ejecución actual.

Esta clase proporciona una serie de métodos estáticos que se delegan en una instancia de SecurityContextHolderStrategy. El propósito de la clase es proporcionar una forma conveniente de especificar la estrategia que debe usarse para una JVM determinada. Esta es una configuración para toda la JVM, ya que todo en esta clase es estático para facilitar el uso en el código de llamada.

Para especificar qué estrategia se debe utilizar, debe proporcionar una configuración de modo. Una configuración de modo es una de las tres configuraciones MODE\_ válidas definidas como campos finales estáticos, o un nombre de clase totalmente calificado para una implementación concreta de SecurityContextHolderStrategy que proporciona un constructor público sin argumentos.

Hay dos formas de especificar el modo de estrategia deseado String. La primera es especificarlo a través de la propiedad del sistema con clave en SYSTEM\_PROPERTY. El segundo es llamar a setStrategyName(String) antes de usar la clase. Si no se utiliza ningún enfoque, la clase utilizará de forma predeterminada MODE\_THREADLOCAL, que es compatible con versiones anteriores, tiene menos incompatibilidades de JVM y es apropiado en los servidores (mientras que MODE\_GLOBAL es definitivamente inapropiado para el uso del servidor).

UsernamePasswordAuthenticationToken

Una implementación de una autorización que se usa para crear un objeto simplemente con el usuario y la contraseña, se crea un objeto Authentication con él, y se almacena en el contexto de Spring. Esto nos permitirá comprobar si el usuario actual está correctamente autenticado.

org.springframework.security.core.userdetails.UserDetailsService

permitirá Interfaz principal que carga datos específicos del usuario.

La interfaz requiere solo un método de solo lectura, lo que simplifica el soporte para nuevas estrategias de acceso a datos:

UserDetails loadUserByUsername (nombre de usuario de cadena) lanza UsernameNotFoundException

Localiza al usuario según el nombre de usuario. En la implementación real, la búsqueda posiblemente distinga entre mayúsculas y minúsculas, o no distinga entre mayúsculas y minúsculas, según cómo esté configurada la instancia de implementación. En este caso, el objeto UserDetails que regresa puede tener un nombre de usuario que sea de un caso diferente al que realmente se solicitó.

Siempre debe devolver un usuario o arrojar excepción, no puede devolver nulo.

JwtAuthenticationFilter

Un filtro de servlet que extrae los tokens web JSON del encabezado de la solicitud de autorización y del parámetro de consulta jwt para usarlos como tokens de autenticación. Si el token trae username y no está guardado en el contexto, en este filtro se almacena.

TokenProvider

Clase con métodos de utilidad donde se puede crear el token, comprobar que es válido, extraer el username…

WebSecurity

Es una clase en la que vamos a personalizar el acceso web a la aplicación. Se indica el tipo de acceso para cada usuario y cada url disponible, lo que se hará con las peticiones no autorizadas, se indican los filtros que pasarán las peticiones, se puede activar la configuración CORS, el filtro Csrf,…

Intercambio de Recursos de Origen Cruzado (CORS) es una característica de seguridad del navegador que restringe las solicitudes HTTP de origen cruzado que se inician desde secuencias de comandos que se ejecutan en el navegador. Si los recursos de la API REST reciben solicitudes HTTP complejas de varios orígenes, debe habilitar la compatibilidad con CORS.

La técnica llamada falsificación de petición en sitios cruzados, proviene de su nombre en inglés Cross Site Request Forgery (CSRF o XSRF). Este ataque fuerza al navegador web de su víctima, validado en algún servicio (como por ejemplo correo o home banking) a enviar una petición a una aplicación web vulnerable.

Esta aplicación se encarga de realizar la acción elegida a través de la víctima, debido que la actividad maliciosa será procesada en nombre del usuario logueado. Al contrario de los ataques conocidos como Cross Site Scripting (su traducción sería ordenes en sitios cruzados – XSS) los cuales explotan la confianza del usuario para con un sitio particular; el Cross Site Request Forgery explota la confianza que un sitio web tiene en un usuario particular.

**MVC**

ResponseEntity

Se usa para dar la respuesta en los Controller de MVC. Se puede incluir el estado (HttpStatus) de la petición, junto al resultado de esta, por ejemplo, un listado, un objeto, …

HttpStatus

Se usa para devolver el código del estado de la petición.

org.springframework.test.web.servlet.MockMvc

Punto de entrada principal para el soporte de prueba Spring MVC del lado del servidor.

**Exceptions**

**public** **class** GlobalExceptionHandler

Es la clase anotada con @RestControllerAdvice donde tenemos todos los métodos que tratarán de forma global las excepciones que se produzcan en la app.

RuntimeException

Es la superclase de aquellas excepciones que se pueden lanzar durante el funcionamiento normal de la máquina virtual Java. RuntimeException y sus subclases son excepciones sin marcar. Las excepciones no verificadas no necesitan ser declaradas en la cláusula throws de un método o constructor si pueden ser lanzadas por la ejecución del método o constructor y propagarse fuera del límite del método o constructor.

ConstraintViolationException

Informa el resultado de violaciones de restricciones.

DataIntegrityViolationException

Es una excepción de datos genéricos que generalmente genera el mecanismo de traducción de excepciones Spring cuando se trata de excepciones de persistencia de nivel inferior.

MissingRequestHeaderException

Indica que un encabezado de solicitud esperado en los parámetros del método de un método @RequestMapping no está presente.

HttpRequestMethodNotSupportedException

Se lanza una excepción cuando un controlador de solicitudes no admite un método de solicitud específico.

AuthenticationException

Superclase abstracta para todas las excepciones relacionadas con un objeto de autenticación que no es válido por cualquier motivo.

MissingServletRequestParameterException

Indica que falta un parámetro.

**Para levantar el contenedor de Docker**

**docker run --network network1 --name SQLSERVERBASEREST -e "ACCEPT\_EULA=Y" -e "SA\_PASSWORD=FranDocker1974" -p 1444:1433 -v C:\sqlserverDocker\baseRest\volumes\data:/var/opt/mssql/data -v C:\sqlserverDocker\baseRest\volumes\log:/var/opt/mssql/log -v C:\sqlserverDocker\baseRest\volumes\secrets:/var/opt/mssql/secrets -d mcr.microsoft.com/mssql/server:2019-CU14-ubuntu-20.04**