Introducción JPA

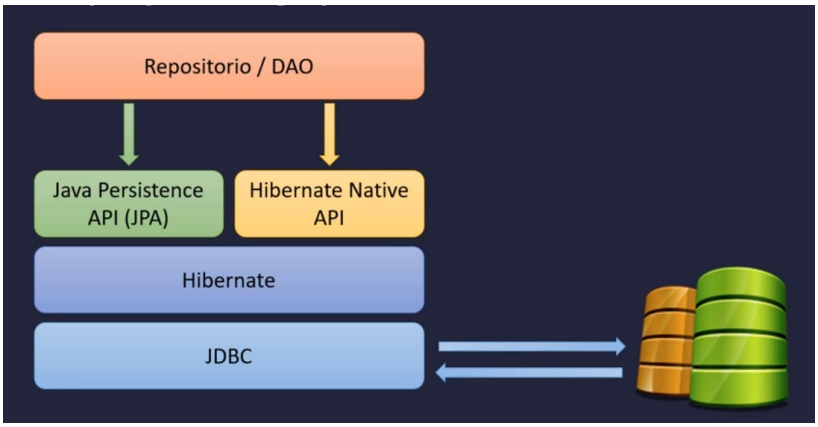
JPA es el estándar, conjunto de interfaces y clases abstractas, que especifica cómo se debe implementar el api de persistencia de Java.

Después tenemos las implementaciones al estándar JPA, por ejemplo hibernate, que implementa y provee el estándar JPA.

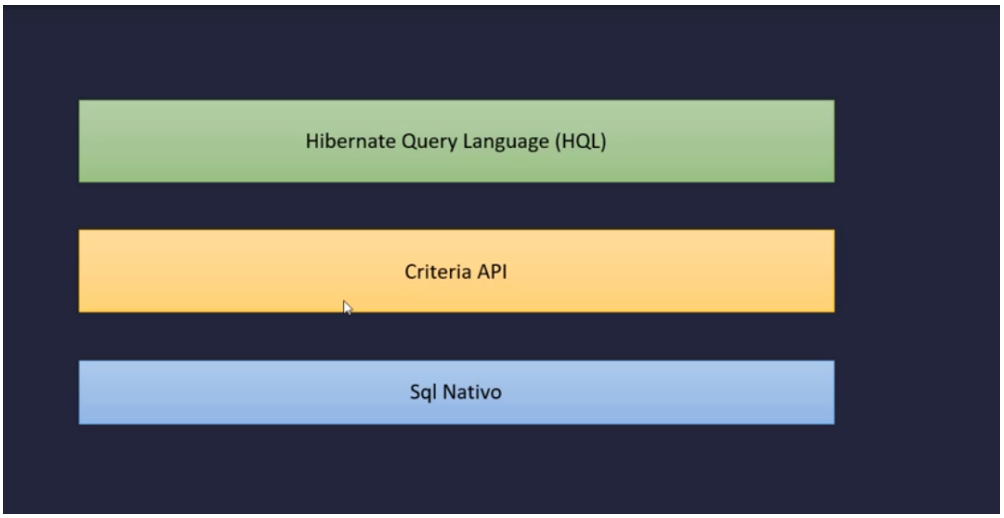
ORM → Objetos mapeados a tablas.

Hibernate

Es una herramienta de mapeo de objeto relacional (ORM) que permite trabajar los datos de una base de datos (RDBMS) en forma de clases y objetos (lenguaje POO).



Tipos de consultas:



HQL:

Consultas sobre las entities:

Select c from cliente c;

Criteria:

Permite construir SQL de forma dinámica, de forma programática.

```
CriteriaQuery<Cliente> criteria = builder.createQuery( Cliente.class
);
Root<Cliente> root = criteria.from( Cliente.class );
criteria.select( root );
criteria.where( builder.equal( root.get( Person_nombre ), "John Doe" )
);
```

Anotaciones.

@NativeQuery

@Query

Documentación:

https://hibernate.org/orm/

https://docs.hibernate.org/stable/orm/userguide/html_single/

Spring JPA:

https://spring.io/projects/spring-data

https://docs.spring.io/spring-data/jpa/reference/

https://docs.spring.io/spring-data/jpa/reference/jpa.html

https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html

Anotaciones Básicas de JPA para la Creación de una Entidad

Java Persistence API (JPA) y su implementación popular, Hibernate, utilizan anotaciones para mapear clases de Java a tablas de bases de datos, facilitando la persistencia de datos en aplicaciones Java. Estas anotaciones permiten definir cómo se almacenan y recuperan los objetos Java de la base de datos sin necesidad de escribir consultas SQL explícitas. A continuación, se describen las anotaciones básicas y algunas importantes para la creación de una entidad en JPA:

- “@Entity”: Indica que una clase es una entidad JPA. Esta anotación se coloca sobre la declaración de clase para informar a JPA que la clase debe ser mapeada a una tabla en la base de datos.

- “@Table”: Especifica la tabla en la base de datos con la que se debe mapear la entidad. Aunque es opcional (si no se especifica, se utiliza el nombre de la clase como nombre de la tabla), es útil para configurar el nombre de la tabla y otros detalles de mapeo de tabla, como el esquema o las restricciones únicas.

```
@Entity
@Table(name = "mi_tabla")
public class MiEntidad { ... }
```

- “@Id”: Marca el campo que actúa como la clave primaria de la entidad. Cada entidad debe tener un campo anotado con “@Id” para identificar de manera única cada instancia de la entidad en la base de datos.

- “@Column”: Se utiliza para especificar el mapeo entre el campo de la entidad y la columna de la base de datos. Permite personalizar el nombre de la columna, su longitud, si puede ser nulo, y otras propiedades de la columna.

```
@Column(name = "nombre", nullable = false, length = 50)
private String nombre;
```

- “@GeneratedValue”: Especifica la estrategia de generación de valores para la clave primaria. Se utiliza junto con “@Id” para indicar cómo se generan los valores de identificación únicos (por ejemplo, autoincremento, secuencia, etc.).

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

Las opciones que acepta strategy son las siguientes:

- GenerationType.IDENTITY: La generación de la clave primaria es delegada a la base de datos. Utiliza una columna auto-incremental o una secuencia en la base de datos para generar los valores únicos de las claves primarias. Es útil cuando se necesita dejar que la base de datos gestione la generación de identificadores.
- GenerationType.SEQUENCE: Utiliza una secuencia de base de datos para generar valores únicos de las claves primarias. Es necesario definir una secuencia en la base de datos y, opcionalmente, se puede usar la anotación @SequenceGenerator para personalizar la secuencia utilizada.
- GenerationType.TABLE: Usa una tabla específica en la base de datos para generar valores únicos de las claves primarias. Este método es más flexible, pero puede ser menos eficiente debido a la necesidad de gestionar una tabla adicional. Normalmente

se usa la anotación @TableGenerator para especificar los detalles de la tabla de generación.

- GenerationType.AUTO: Permite que el proveedor de persistencia (como Hibernate) elija la estrategia de generación más adecuada según la base de datos en uso. Es el valor por defecto y proporciona una estrategia de generación automática.

- "@Enumerated": Se utiliza para mapear un campo enum a una columna de la base de datos. Permite especificar si el mapeo debe usar el nombre del enum (String) o su ordinal (número).

- @Temporal: Se utiliza para mapear campos de fecha y hora ("java.util.Date" o "java.util.Calendar") a una columna de base de datos, especificando el tipo de fecha/hora (fecha, hora, o fecha y hora).

Clase entidad:

```
@Entity
public class Cliente {

@Id
private Long id;
private String nombre;
private String apellido;
@Column(name="forma_pago")
private Date formaPago;
//Getters y setters
```

Si la anotación `@Entity` no tiene la propiedad table se une a una tabla con el mismo nombre "Cliente".

La anotación `@id` para crear una clave primaria, podríamos cambiarle el nombre con `@Column`

Si no anotamos los atributos se crearán de manera básica.

Ciclo de vida de las entidades.

@PrePersist

La anotación "@PrePersist" en Spring se utiliza en el contexto de JPA (Java Persistence API), que es un estándar para la persistencia de datos y la creación de ORM (Object-Relational Mapping) en Java.

"@PrePersist" se utiliza para configurar un método de devolución de llamada que se ejecutará justo antes de que la entidad correspondiente se inserte por primera vez en la base de datos. Es útil para realizar ciertas operaciones justo antes de la persistencia de la entidad, como establecer valores predeterminados o registrar la fecha y hora de creación.

Veamos un ejemplo de cómo se puede utilizar:

```
import javax.persistence.PrePersist;
import java.time.LocalDateTime;
```

```
@Entity
public class User {
    // ...

    private LocalDateTime createdAt;

    @PrePersist
    public void prePersist() {
        this.createdAt = LocalDateTime.now();
    }

    // ...
}
```

En este ejemplo, el método “prePersist” se ejecutará justo antes de que se persista una nueva entidad “User”. Establece el campo “createdAt” en la fecha y hora actuales. De esta manera, cada vez que crees un nuevo “User”, su fecha de creación se registrará automáticamente.

@PostPersist

La anotación “@PostPersist” en Spring se utiliza en el contexto de JPA (Java Persistence API). Al igual que “@PrePersist”, “@PostPersist” se utiliza para configurar un método de devolución de llamada, pero este se ejecuta justo después de que la entidad correspondiente se haya insertado con éxito en la base de datos.

Esto puede ser útil para realizar ciertas operaciones que dependen de la creación exitosa de la entidad, como el registro de eventos o la notificación a otros componentes de la aplicación.

Veamos un ejemplo de cómo se puede utilizar:

```
import javax.persistence.PostPersist;

@Entity
public class User {
    // ...

    @PostPersist
    public void postPersist() {
        System.out.println("User has been created with id: " +
this.id);
    }

    // ...
}
```

En este ejemplo, el método “postPersist” se ejecutará justo después de que se persista una nueva entidad “User”. Imprime un mensaje indicando que el usuario ha sido creado, incluyendo su id. Esto podría ser útil para fines de depuración o registro.

@Embedded

La anotación “@Embedded” en Spring se utiliza en el contexto de JPA (Java Persistence API) para indicar que una entidad tiene un atributo que es una clase incrustada.

Una clase incrustada es una clase que no es una entidad por sí misma, pero cuyos atributos se mapean directamente a la tabla de la entidad que la contiene. Esto puede ser útil para agrupar atributos relacionados juntos en su propia clase, lo que puede hacer que tu código sea más organizado y fácil de entender.

Veamos un ejemplo de cómo se puede utilizar:

```
import javax.persistence.Embeddable;
import javax.persistence.Embedded;

@Entity
public class User {
    // ...

    @Embedded
    private Address address;

    // ...
}

@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    private String zip;

    // getters and setters
}
```

En este ejemplo, “User” tiene un atributo “address” que es una instancia de la clase “Address”. “Address” está anotada con “@Embeddable” para indicar que es una clase incrustada, y el atributo “address” en “User” está anotado con “@Embedded” para indicar que es un atributo incrustado.

Como resultado, los atributos de “Address” (street, city, state, zip) se mapearán directamente a la tabla “User” en la base de datos.

Aplication.properties

El fichero “application.properties” en Spring es un archivo de configuración centralizado utilizado para definir propiedades y configuraciones para una aplicación Spring Boot. Este archivo permite a los desarrolladores personalizar el comportamiento de su aplicación, incluyendo la configuración del servidor, parámetros de base de datos, niveles de logging, y más. Spring Boot utiliza estos valores para configurar automáticamente la aplicación durante el arranque.

Ubicación

Por defecto, el archivo “application.properties” se ubica en el directorio “src/main/resources” de un proyecto Spring Boot. Esta ubicación permite que el archivo sea empaquetado dentro del artefacto construido (por ejemplo, un archivo JAR o WAR), haciendo que las configuraciones estén disponibles en el entorno de ejecución.

Funcionalidades Clave

- Configuración de Base de Datos: Permite especificar detalles de conexión a la base de datos, como URL, nombre de usuario, contraseña, y configuraciones de pool de conexiones.

```
spring.datasource.url=jdbc:mysql://localhost:3306/mi_base_de_datos
spring.datasource.username=usuario
spring.datasource.password=contraseña
```

- Configuración del Servidor: Define propiedades del servidor embebido, como el puerto en el que se ejecuta la aplicación o el contexto de la aplicación.

```
server.port=8080
server.servlet.context-path=/miapp
```

- Configuración de Logging: Personaliza los niveles de logging de la aplicación y de las dependencias.

```
logging.level.org.springframework=INFO
logging.level.com.miapp=DEBUG
 logging.file.name=app.log
```

- Configuración de JPA/Hibernate: Configura aspectos de JPA y Hibernate, como el dialecto de la base de datos o la generación de DDL.

```
spring.jpa.hibernate.ddl-auto=update
```

La propiedad acepta los siguientes valores:

- none: No realiza ninguna acción en el esquema de la base de datos. Hibernate no valida ni genera el esquema.
- validate: Hibernate valida el esquema de la base de datos comparando las entidades mapeadas con la estructura de la base de datos. Si hay discrepancias, se lanzará un error.
- update: Hibernate actualiza el esquema de la base de datos, aplicando los cambios necesarios para que coincidan con las entidades mapeadas. Es útil en desarrollo, pero debe usarse con precaución en producción.
- create: Hibernate crea el esquema de la base de datos desde cero cada vez que se inicia la aplicación. Esto implica que se eliminarán todos los datos existentes.
- create-drop: Similar a create, pero Hibernate también elimina el esquema de la base de datos cuando la sesión de la fábrica de entidades (EntityManagerFactory) se cierra. Es útil para pruebas rápidas.

- drop: Hibernate elimina el esquema de la base de datos cuando la sesión de la fábrica de entidades se cierra.

- Configuraciones Específicas de la Aplicación: Además de las configuraciones predeterminadas de Spring Boot, los desarrolladores pueden definir sus propias propiedades personalizadas para ser utilizadas dentro de la aplicación.

```
miapp.configuracion.especial=valorEspecial
```

Ventajas

- Centralización de Configuraciones: Facilita la gestión de configuraciones al mantenerlas en un único lugar.

- Flexibilidad: Permite sobrescribir configuraciones para diferentes entornos (desarrollo, prueba, producción) utilizando perfiles de Spring (“application-{profile}.properties”).

- Automatización: Spring Boot utiliza estas configuraciones para autoconfigurar componentes de la aplicación, reduciendo la necesidad de configuración manual y código boilerplate.

El archivo “application.properties” ayuda a los desarrolladores a gestionar las configuraciones de su aplicación de manera eficiente y flexible, adaptándose a las necesidades específicas de cada proyecto.

Ejemplo MySQL:

```
spring.datasource.url=jdbc:mysql://localhost:3306/appPeliculas
spring.datasource.username=appPeliculas
spring.datasource.password=1234
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDia
lect
```

H2

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.defer-datasource-initialization=true
spring.jpa.show-sql=true
spring.h2.console.enabled=true
```

Datos de ejemplo:

Crear un archivo import.sql en resources, al arrancar hace el insert, siempre que `spring.jpa.hibernate.ddl-auto=create`

Asociaciones:

- @OneToOne
- @ManyToOne
- @OneToMany
- @OneToOne

@OneToOne

Para explicar "@OneToOne" sin bidireccionalidad, consideremos un ejemplo simple donde cada usuario tiene asignado exactamente un perfil. En este caso, no necesitamos navegar desde "Perfil" a "Usuario", solo de "Usuario" a "Perfil".

Veamos un ejemplo de Relación "@OneToOne"

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

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "perfil_id", referencedColumnName = "id")
    private Perfil perfil;

    // Getters y setters...
}

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

    private String detalles;

    // Getters y setters...
}
```

En este ejemplo:

- La entidad "Usuario" tiene un campo "perfil" que es una referencia a la entidad "Perfil". La anotación "@OneToOne" indica que se trata de una relación uno a uno. La anotación "@JoinColumn" especifica la columna que se utilizará como clave foránea ("perfil_id") para establecer la relación con "Perfil".

- La entidad "Perfil" no tiene referencia a "Usuario", lo que significa que la relación es unidireccional. Solo puedes navegar de "Usuario" a "Perfil", pero no al revés.

Este modelo es adecuado cuando tienes una clara relación uno a uno entre dos entidades, pero solo necesitas acceder a una de las entidades desde la otra, no necesitando la navegación inversa.

@ManyToOne

La anotación "@ManyToOne" en Spring JPA se utiliza para establecer una relación de muchos a uno entre dos entidades.

En una relación de muchos a uno, muchas entidades de un tipo pueden estar asociadas con una única entidad de otro tipo. Por ejemplo, podrías tener una entidad "Order" y una entidad "Customer", donde cada "Order" está asociada a un "Customer", pero un "Customer" puede tener muchas "Order".

Veamos un ejemplo de cómo se puede utilizar:

```
import javax.persistence.ManyToOne;

@Entity
public class Order {
    // ...

    @ManyToOne
    private Customer customer;

    // ...
}

@Entity
public class Customer {
    // ...
}
```

En este ejemplo, cada "Order" está asociada a un "Customer". Cuando se guarda una "Order", se guarda una referencia al "Customer" asociado en la base de datos.

Puedes usar "@ManyToOne" junto con "@JoinColumn" para personalizar la columna que se utiliza para almacenar la referencia al "Customer".

```
@ManyToOne
@JoinColumn(name="customer_id")
private Customer customer;
```

En este caso, la referencia al "Customer" se almacenará en una columna llamada "customer_id" en la tabla "Order".

@OneToMany

(crea tabla intermedia.)

La anotación “@OneToMany” en Spring JPA se utiliza para establecer una relación de uno a muchos entre dos entidades.

En una relación de uno a muchos, una entidad de un tipo puede estar asociada con muchas entidades de otro tipo. Por ejemplo, podrías tener una entidad “Customer” y una entidad “Order”, donde un “Customer” puede tener muchas “Order”, pero cada “Order” está asociada a un solo “Customer”.

Veamos un ejemplo de cómo se puede utilizar:

```
import javax.persistence.OneToMany;

@Entity
public class Customer {
    // ...

    @OneToMany(mappedBy = "customer")
    private List<Order> orders;

    // ...
}

@Entity
public class Order {
    // ...

    @ManyToOne
    private Customer customer;

    // ...
}
```

En este ejemplo, cada “Customer” puede tener muchas “Order”. La anotación “@OneToMany” en el campo “orders” en “Customer” establece esta relación. El atributo “mappedBy” indica el campo en la entidad “Order” que mantiene la relación inversa.

Por otro lado, cada “Order” está asociada a un “Customer”, como se indica con la anotación “@ManyToOne” en el campo “customer” en “Order”.

Cuando se guarda un “Customer” con una lista de “Order”, JPA actualizará la base de datos de manera que cada “Order” en la lista apunte a la “Customer” correspondiente.

@ManyToMany

La anotación “@ManyToMany” en JPA también se puede utilizar para definir relaciones de muchos a muchos sin bidireccionalidad, es decir, cuando solo una de las entidades necesita conocer la existencia de la otra. Esto puede ser útil para simplificar el modelo de dominio cuando no necesitas navegar la relación en ambas direcciones.

Veamos un ejemplo de una relación “@ManyToMany” sin bidireccionalidad:

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

```
    // Otros atributos...

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();

    // Getters y setters...
}

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

    // Otros atributos...

    // No hay referencia a Student desde Course

    // Getters y setters...
}
```

En este ejemplo, la entidad “Student” tiene una relación “@ManyToMany” con “Course”, pero la entidad “Course” no tiene conocimiento de “Student”. La relación se configura de la misma manera que en el ejemplo bidireccional, utilizando una tabla de unión especificada por la anotación “@JoinTable”. Sin embargo, en este caso, no se utiliza “mappedBy” en la entidad “Course” porque no estamos estableciendo una relación bidireccional.

Este enfoque es útil cuando solo necesitas acceder a los cursos de un estudiante, pero no necesitas acceder a los estudiantes de un curso directamente desde la entidad “Course”. Reducir la bidireccionalidad en las relaciones puede simplificar el modelo de dominio y mejorar el rendimiento al evitar cargas innecesarias de datos relacionados.

Cascade

En JPA, la propiedad “cascade” se utiliza para propagar las operaciones de la entidad padre a las entidades hijo. Esto es útil en las relaciones entre entidades, como “@OneToOne”, “@OneToMany”, “@ManyToOne” y “@ManyToMany”.

Las operaciones en cascada disponibles son:

- “CascadeType.PERSIST”: Si la entidad padre se persiste, la entidad hijo también se persistirá.
- “CascadeType.REMOVE”: Si la entidad padre se elimina, la entidad hijo también se eliminará.
- “CascadeType.MERGE”: Si la entidad padre se fusiona, la entidad hijo también se fusionará.
- “CascadeType.REFRESH”: Si la entidad padre se actualiza, la entidad hijo también se actualizará.
- “CascadeType.DETACH”: Si la entidad padre se desconecta, la entidad hijo también se

desconectará.

- "CascadeType.ALL": Todas las operaciones anteriores se aplicarán.

Veamos un ejemplo de cómo se puede utilizar:

```
@Entity
public class Post {
    // ...

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL)
    private List<Comment> comments = new ArrayList<>();

    // ...
}

@Entity
public class Comment {
    // ...

    @ManyToOne
    private Post post;

    // ...
}
```

En este ejemplo, cada "Post" puede tener muchos "Comment". La anotación "@OneToMany" en el campo "comments" en "Post" establece esta relación. La propiedad "cascade = CascadeType.ALL" indica que cualquier operación realizada en "Post" se propagará a sus "Comment" asociados. Por ejemplo, si un "Post" se elimina, todos sus "Comment" asociados también se eliminarán.

orphanRemoval

La propiedad "orphanRemoval" en JPA se utiliza para especificar que las entidades hijo que ya no están asociadas a la entidad padre deben eliminarse. Es útil en las relaciones "@OneToOne" y "@OneToMany".

Cuando "orphanRemoval" está establecido en "true", si quitas la relación entre la entidad padre y la entidad hijo (por ejemplo, quitas un objeto de una colección en la entidad padre), la entidad hijo se eliminará automáticamente de la base de datos. Esto es útil cuando la entidad hijo no tiene sentido sin la entidad padre.

Veamos un ejemplo de cómo se puede utilizar:

```
@Entity
public class Cart {
    // ...

    @OneToMany(mappedBy = "cart", orphanRemoval = true)
    private List<Item> items = new ArrayList<>();

    // ...
}
```

```
@Entity
public class Item {
    // ...

    @ManyToOne
    private Cart cart;

    // ...
}
```

En este ejemplo, cada “Cart” puede tener muchos “Item”. La anotación “@OneToMany” en el campo “items” en “Cart” establece esta relación. La propiedad “orphanRemoval = true” indica que si un “Item” se quita de la “Cart”, se eliminará automáticamente de la base de datos. Esto tiene sentido en este contexto, ya que un “Item” que no está en una “Cart” ya no es relevante y puede ser eliminado.

Bidireccionalidad

La bidireccionalidad en Hibernate y Spring JPA se refiere a la capacidad de navegar una relación entre dos entidades en ambas direcciones. En una relación bidireccional, cada entidad tiene una referencia a la otra, permitiendo el acceso a los datos relacionados desde cualquiera de los dos lados de la relación.

Las relaciones bidireccionales pueden ser de varios tipos, incluyendo “@OneToOne”, “@OneToMany”/”@ManyToOne”, y “@ManyToMany”. La bidireccionalidad se establece mediante el uso de anotaciones en ambos lados de la relación, con una de las entidades actuando como el lado "propietario" de la relación (el que maneja la actualización de la relación en la base de datos) y la otra como el lado "inverso" o "mapeado por".

Ejemplo de “@OneToMany” y “@ManyToOne” (Relación Bidireccional)

Considera una relación entre “Post” y “Comment” donde un “Post” puede tener muchos “Comment”, pero un “Comment” pertenece a un solo “Post”.

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

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Comment> comments = new ArrayList<>();

    // Getters y setters...
}
```

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

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters y setters...
}
```

- **Lado Propietario (“Comment”)**: El lado propietario es donde defines la columna de clave foránea (“post_id”). Aquí, “Comment” es el propietario de la relación porque contiene la columna de clave foránea que referencia a “Post”.

- **Lado Inverso (“Post”)**: El lado inverso es donde usas “mappedBy” para indicar que la relación es manejada por el otro lado. En este caso, “Post” es el lado inverso, indicando que la relación está mapeada por la propiedad “post” en “Comment”.

Beneficios de la Bidireccionalidad

- **Navegación**: Permite navegar la relación en ambas direcciones, desde “Post” a “Comment” y viceversa.

- **Sincronización**: Ayuda a mantener sincronizadas las dos partes de la relación, asegurando la integridad de los datos.

- **Eficiencia en Consultas**: Puede mejorar la eficiencia de las consultas al permitir el acceso directo a los datos relacionados desde cualquiera de los lados de la relación.

Consideraciones

- **Manejo de Lado Propietario**: Es importante manejar correctamente el lado propietario de la relación para asegurar que las actualizaciones en la base de datos reflejen el estado de los objetos en memoria.

- **Rendimiento**: Las relaciones bidireccionales pueden afectar el rendimiento si no se usan con cuidado, especialmente si se cargan grandes volúmenes de datos relacionados innecesariamente.

La bidireccionalidad es una buena herramienta en JPA y Hibernate, pero requiere un diseño cuidadoso y una gestión adecuada para aprovechar sus beneficios sin incurrir en problemas de rendimiento o mantenimiento.

Otro ejemplo con Bidireccionalidad.

Una relación “@OneToOne” en JPA (Java Persistence API) se utiliza para modelar una relación en la que una entidad está asociada con exactamente una instancia de otra entidad. Por ejemplo, considera una situación donde cada persona tiene asignado un único pasaporte, y

cada pasaporte está asociado con una única persona. Esta es una relación uno a uno entre “Persona” y “Pasaporte”.

Para implementar una relación “@OneToOne” en JPA, puedes usar la anotación “@OneToOne”. Aquí hay un ejemplo que muestra cómo hacerlo:

Ejemplo de Relación “@OneToOne”

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

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "pasaporte_id", referencedColumnName = "id")
    private Pasaporte pasaporte;

    // Getters y setters...
}

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

    private String numero;

    @OneToOne(mappedBy = "pasaporte")
    private Persona persona;

    // Getters y setters...
}
```

En este ejemplo:

- La entidad “Persona” tiene un campo “pasaporte” que es una referencia a la entidad “Pasaporte”. La anotación “@OneToOne” indica que se trata de una relación uno a uno. La anotación “@JoinColumn” especifica la columna que se utilizará como clave foránea para establecer la relación.

- La entidad “Pasaporte” tiene un campo “persona” que referencia a la entidad “Persona”. La anotación “@OneToOne(mappedBy = "pasaporte")” indica que esta es la otra parte de una relación uno a uno bidireccional y que el mapeo es manejado por el campo “pasaporte” en la entidad “Persona”.

Consideraciones

- Bidireccionalidad: En el ejemplo, la relación es bidireccional, lo que significa que puedes navegar desde “Persona” a “Pasaporte” y viceversa. La bidireccionalidad se establece mediante el uso de “mappedBy” en el lado no propietario (“Pasaporte”).

- Cascada: La opción “cascade = CascadeType.ALL” en “Persona” indica que las operaciones como persistir, eliminar, etc., aplicadas a “Persona” también deben aplicarse a “Pasaporte”. Esto es útil para mantener sincronizadas las operaciones entre las entidades relacionadas.

- Clave Foránea: La relación “@OneToOne” generalmente implica una restricción de clave foránea en la base de datos, asegurando la integridad referencial entre las dos tablas.

Este modelo es útil cuando necesitas asegurar que cada entidad en una relación esté vinculada a una y solo una instancia de la otra entidad, como en el caso de personas y sus pasaportes.

Relación ManyToMany con atributo en la relación intermedia

Para añadir un atributo en la tabla intermedia de una relación bidireccional ManyToMany con JPA, puedes seguir estos pasos. En este ejemplo, añadiremos un atributo “fecha” a la tabla intermedia:

1.- Crear una entidad para la tabla intermedia

```
@Entity
public class EntidadIntermedia {

    @EmbeddedId
    private EntidadIntermediaId id;

    @ManyToOne
    @MapsId("entidadAId")
    @JoinColumn(name = "entidad_a_id") private EntidadA entidadA;

    @ManyToOne
    @MapsId("entidadBId")
    @JoinColumn(name = "entidad_b_id")
    private EntidadB entidadB;

    private LocalDate fecha;

    // getters and setters
}
```

2.- Crear una clase embebida para el ID de la entidad intermedia

```
@Embeddable
public class EntidadIntermediaId implements Serializable {

    private Long entidadAId; private Long entidadBId;

    // getters, setters, equals y hashCode
}
```

3.- Actualizar las entidades originales para incluir la relación

```
@Entity
public class EntidadA {

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

    @OneToMany(mappedBy = "entidadA", cascade = CascadeType.ALL) private Set<EntidadIntermedia> entidadIntermediaSet = new HashSet<>();

    // getters y setters
}

@Entity
public class EntidadB {

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

    @OneToMany(mappedBy = "entidadB", cascade = CascadeType.ALL) private Set<EntidadIntermedia> entidadIntermediaSet = new HashSet<>();

    // getters y setters
}
```

Tipos de búsqueda o fetching:

En JPA (Java Persistence API), que es ampliamente utilizado por Hibernate y Spring Data JPA, el "fetching" se refiere a la estrategia utilizada para cargar las relaciones de una entidad desde la base de datos. Hay dos tipos principales de estrategias de fetching: Eager (Ansioso) y Lazy (Perezoso).

Fetch Eager (Carga Ansiosa)

- **Definición**: En la estrategia de carga ansiosa, las relaciones de una entidad se cargan inmediatamente con la entidad principal, incluso si no se necesitan. Esto significa que cuando

accedes a la entidad principal, todas sus relaciones configuradas con Fetch Eager también se recuperan de la base de datos en la misma consulta o en consultas inmediatas adicionales.

- **Uso**: Se utiliza cuando sabes que vas a necesitar las relaciones de la entidad cada vez que accedas a ella. Es útil cuando las relaciones son esenciales para las operaciones que estás realizando y cuando el tamaño del conjunto de datos relacionados es pequeño o mediano.

- **Configuración**: Se configura usando la anotación "@Fetch(FetchMode.JOIN)" o "@ManyToOne(fetch = FetchType.EAGER)" y "@OneToMany(fetch = FetchType.EAGER)".

```
@Entity
public class Libro {
    @ManyToOne(fetch = FetchType.EAGER)
    private Autor autor;
}
```

Fetch Lazy (Carga Perezosa)

- **Definición**: En la estrategia de carga perezosa, las relaciones de una entidad no se cargan cuando se carga la entidad principal; en su lugar, se cargan solo cuando se accede a ellas por primera vez. Esto puede mejorar el rendimiento al evitar la carga de datos innecesarios.

- **Uso**: Se utiliza cuando las relaciones de una entidad no siempre son necesarias. Es útil para mejorar el rendimiento, especialmente en casos donde hay grandes conjuntos de datos o cuando la carga de todas las relaciones de una sola vez podría ser costosa en términos de recursos.

- **Configuración**: Se configura usando la anotación "@ManyToOne(fetch = FetchType.LAZY)" y "@OneToMany(fetch = FetchType.LAZY)".

```
@Entity
public class Libro {
    @ManyToOne(fetch = FetchType.LAZY)
    private Autor autor;
}
```

Consideraciones

- Lazy Initialization Exception: Al usar la carga perezosa, puedes encontrarte con una "LazyInitializationException" si intentas acceder a una relación perezosa después de que la sesión o el contexto de persistencia se haya cerrado. Esto se debe a que la entidad intenta cargar la relación fuera del alcance de una sesión abierta.

-Uso de Entity Graphs: Para casos específicos donde necesitas un control más fino sobre qué relaciones cargar, puedes usar Entity Graphs en JPA 2.1 para especificar dinámicamente qué relaciones cargar, independientemente de la configuración de fetching por defecto.

La elección entre Fetch Eager y Fetch Lazy depende de las necesidades específicas de tu aplicación, el modelo de datos y los patrones de acceso a los datos. Un uso cuidadoso de estas estrategias puede mejorar significativamente el rendimiento de tu aplicación.

Tipos de búsquedas (fetching) por defecto:

En JPA (Java Persistence API), el comportamiento predeterminado de fetching (carga) para las diferentes relaciones entre entidades (@OneToOne, @OneToMany, @ManyToOne, @ManyToMany) varía. Este comportamiento predeterminado puede ser sobrescrito utilizando la propiedad "fetch" en las anotaciones de relación.

@OneToOne

- **Fetch Predeterminado**: EAGER (Ansioso)

- **Razón**: Por defecto, JPA asume que si tienes una relación uno a uno, probablemente necesitarás acceder a la entidad relacionada cada vez que accedas a la entidad propietaria. Por lo tanto, carga la entidad relacionada de manera ansiosa.

- Ejemplo:

```
@OneToOne
private DetalleUsuario detalleUsuario;
```

@OneToMany

- **Fetch Predeterminado**: LAZY (Perezoso)

- **Razón**: Las relaciones uno a muchos pueden potencialmente involucrar la carga de un gran número de entidades relacionadas. Para evitar un impacto negativo en el rendimiento debido a la carga de grandes volúmenes de datos que quizás no se necesiten, JPA opta por un enfoque perezoso por defecto.

- Ejemplo

```
@OneToMany(mappedBy = "usuario")
private List<Publicacion> publicaciones;
```

@ManyToOne

- **Fetch Predeterminado**: EAGER (Ansioso)

- **Razón**: Similar a @OneToOne, la suposición es que es más probable que necesites acceder a la entidad relacionada cada vez que consultes la entidad propietaria. Por lo tanto, JPA prefiere cargar estos datos de manera ansiosa por defecto.

- Ejemplo:

```
@ManyToOne
private Usuario usuario;
```

@ManyToMany

- **Fetch Predeterminado**: LAZY (Perezoso)

- **Razón**: Las relaciones muchos a muchos pueden ser especialmente costosas en términos de rendimiento, ya que involucran la carga de múltiples entidades de ambos lados de la relación. Para minimizar el impacto en el rendimiento, JPA utiliza la carga perezosa por defecto.

- Ejemplo:

```
@ManyToMany
private List<Curso> cursos;
```

Consideraciones Importantes

- **Sobrescribir el Comportamiento Predeterminado**: Puedes sobrescribir el comportamiento de fetching predeterminado utilizando la propiedad “fetch” en las anotaciones de relación. Por ejemplo, “@OneToMany(fetch = FetchType.EAGER)” cambiará una relación uno a muchos a carga ansiosa.

- **Impacto en el Rendimiento**: Elegir entre carga ansiosa y perezosa tiene implicaciones significativas en el rendimiento de tu aplicación. La carga ansiosa puede llevar a la sobre-carga de datos innecesarios, mientras que la carga perezosa puede resultar en un mayor número de consultas a la base de datos.

- **Manejo de LazyInitializationException**: Al usar carga perezosa, es común encontrarse con “LazyInitializationException” si se accede a la entidad relacionada fuera del contexto de una transacción abierta. Es importante gestionar estas situaciones, por ejemplo, utilizando patrones como Open Session in View o asegurándose de acceder a las relaciones dentro de los límites de una transacción.

La elección entre carga ansiosa y perezosa debe basarse en el conocimiento específico de tus necesidades de acceso a datos y patrones de uso para optimizar el rendimiento y la eficiencia de tu aplicación.

N+1 consultas:

El problema de las consultas N + 1 es un problema común de rendimiento en aplicaciones que utilizan Hibernate o JPA (Java Persistence API) para acceder a datos en una base de datos relacional. Este problema ocurre cuando se ejecuta una consulta para obtener una lista de entidades, y luego, para cada entidad recuperada, se realiza una consulta adicional para recuperar datos relacionados. Esto resulta en una consulta inicial (la "1" en "N + 1") más N consultas adicionales, donde N es el número de entidades recuperadas por la consulta inicial.

Ejemplo del Problema N + 1

Imagina que tienes una entidad “Libro” y una entidad “Autor”, donde cada “Libro” tiene una referencia a un “Autor”. Si quieres recuperar una lista de libros y mostrar el autor de cada libro, podrías hacer algo como esto:

```
@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {
    List<Libro> findAll();
}

@Service
public class LibroService {
    @Autowired
    private LibroRepository libroRepository;

    public void imprimirAutoresDeLibros() {
        List<Libro> libros = libroRepository.findAll();
        for (Libro libro : libros) {
            System.out.println(libro.getAutor().getNombre());
        }
    }
}
```

Si tienes 10 libros en tu base de datos, esta operación podría resultar en 11 consultas a la base de datos:

1.- Una consulta para recuperar todos los libros.
2.- Una consulta adicional para cada libro para recuperar su autor.

Por qué es un Problema

El problema de las consultas N + 1 puede degradar significativamente el rendimiento de una aplicación, especialmente cuando N es grande. Esto se debe a que cada consulta adicional incurre en costos de red y procesamiento, lo que puede sumarse rápidamente.

Cómo Resolver el Problema

Para evitar el problema de las consultas N + 1, puedes utilizar técnicas como:

- **Fetch Join**: Puedes utilizar una cláusula JOIN FETCH en tu consulta JPQL o HQL para recuperar la entidad principal y sus relaciones en una sola consulta.

```
@Query("Select l from Libro l JOIN FETCH l.autor")
```

```
Optional<Libro> findOne(Long id);
```

- **Entity Graphs**: JPA 2.1 introdujo los entity graphs que permiten especificar de manera dinámica qué relaciones deben ser precargadas con la entidad principal.

```
@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {
    @EntityGraph(attributePaths = {"autor"})
    List<Libro> findAll();
}
```

Conclusión

Para evitar el problema de las consultas N + 1 se debe tener un diseño cuidadoso y el uso de técnicas adecuadas para la recuperación de datos, ya que esto minimizará o evitará su impacto en el rendimiento de la aplicación.

Anotación @Transaccional

La anotación "@Transactional" en Spring Framework es utilizada para declarar que un método o clase entera debe ser ejecutado dentro de un contexto de transacción. Cuando se utiliza con Spring Data JPA, esta anotación maneja la apertura, el commit o el rollback de transacciones de manera declarativa, simplificando el manejo de transacciones en aplicaciones que acceden a bases de datos.

Funcionamiento Básico

- **Apertura de Transacción**: Al entrar a un método anotado con "@Transactional", Spring verifica si ya existe una transacción activa. Si no hay ninguna, Spring inicia una nueva transacción.

- **Commit/Rollback**: Si el método se ejecuta sin lanzar excepciones, Spring intenta hacer commit de la transacción, aplicando todos los cambios realizados durante la transacción a la base de datos. Si ocurre una excepción, Spring realiza un rollback de la transacción, deshaciendo todos los cambios realizados en la base de datos durante esa transacción.

- **Propagación de Transacciones**: La anotación permite especificar un comportamiento de propagación, que determina cómo las transacciones se relacionan entre sí. Por ejemplo, si un método anotado con "@Transactional" es llamado por otro método que ya está dentro de una transacción, puedes configurar si quieres que el método use la transacción existente o inicie una nueva.

- **Aislamiento de Transacciones**: También puedes especificar el nivel de aislamiento de la transacción, que define cómo una transacción debe ser aislada de otras transacciones en términos de acceso a los datos.

Ejemplo de Uso

```
import org.springframework.stereotype.Service;
```

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

@Service
public class MiServicio {

    @Transactional
    public void metodoDeServicio() {
        // Lógica de negocio que accede a la base de datos.
        // Cualquier cambio en la base de datos aquí será parte de una
transacción.
    }
}
```

Consideraciones

- **Manejo de Excepciones**: Por defecto, "@Transactional" solo marca para rollback las excepciones de tiempo de ejecución ("RuntimeException" y sus subclases). Si deseas que también se haga rollback para excepciones chequeadas, necesitas configurarlo explícitamente usando el atributo "rollbackFor".

- **Proxy**: Spring implementa "@Transactional" mediante el uso de proxies. Esto significa que la anotación solo se aplica cuando el método anotado es llamado desde fuera de la clase. Llamar a un método "@Transactional" desde otro método de la misma instancia de clase no aplicará el manejo de transacciones de Spring.

- **Integración con JPA**: "@Transactional" en combinación con Spring Data JPA maneja automáticamente la persistencia del contexto (EntityManager) y asegura que las operaciones sobre la base de datos sean parte de la transacción declarada.

La anotación "@Transactional" es una potente herramienta en Spring para el manejo declarativo de transacciones, permitiendo a los desarrolladores enfocarse en la lógica de negocio sin preocuparse por los detalles de bajo nivel del manejo de transacciones.

Recursión infinita en entidades bidireccionales.

Al utilizar los métodos "toString()" generados automáticamente por el IDE sin personalización adecuada, se produce una excepción que se conoce como "StackOverflowError" debido a la recursión infinita.

¿Cómo ocurre?

Cuando tienes dos entidades con una relación bidireccional (por ejemplo, "@OneToMany" y "@ManyToOne" entre "ClaseA" y "ClaseB"), y ambas entidades incluyen referencias a la otra en sus métodos "toString()", intentar imprimir una instancia de cualquiera de estas entidades resultará en una llamada recursiva infinita entre estos métodos "toString()". Esto sucede porque:

1.- Al intentar imprimir "ClaseA", su método "toString()" intenta imprimir también la información de "ClaseB" asociada.
2.- Al imprimir "ClaseB", su método "toString()" intenta imprimir la información de "ClaseA" asociada.

3.- Este ciclo continúa indefinidamente, ya que cada entidad intenta imprimir la otra, llevando eventualmente a un desbordamiento de la pila de llamadas ("StackOverflowError").

Ejemplo:

```
class ClaseA {
    @OneToOne
    private ClaseB claseB;

    @Override
    public String toString() {
        return "ClaseA{" + "claseB=" + claseB + '}';
    }
}

class ClaseB {
    @OneToOne
    private ClaseA claseA;

    @Override
    public String toString() {
        return "ClaseB{" + "claseA=" + claseA + '}';
    }
}
```

Intentar imprimir una instancia de "ClaseA" o "ClaseB" resultará en un "StackOverflowError".

Soluciones:

Para evitar este problema, puedes:

- Personalizar los métodos "toString()": Modifica los métodos "toString()" para que no incluyan relaciones bidireccionales directamente, o maneja la impresión de manera condicional para evitar la recursión.

- Usar "@ToString.Exclude" con Lombok: Si estás utilizando Lombok, puedes excluir las relaciones bidireccionales en tus métodos "toString()" usando la anotación "@ToString.Exclude".

- Implementar lógica de control: Añade lógica para controlar la profundidad de la recursión o para detectar y evitar la inclusión de referencias ya impresas.

Ejemplo solucionado:

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

    @OneToOne(mappedBy = "claseA")
    private ClaseB claseB;

    // Getters y Setters

    @Override
    public String toString() {
```

```
        // Implementación personalizada para evitar StackOverflowError
        return "ClaseA{" + "id=" + id + '}';
    }
}

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

    @OneToOne
    private ClaseA claseA;

    // Getters y Setters

    @Override
    public String toString() {
        // Implementación personalizada para evitar StackOverflowError
        return "ClaseB{" + "id=" + id + '}';
    }
}
```

Evitar la recursión infinita en los métodos “toString()” es crucial para prevenir “StackOverflowError” y asegurar que tu aplicación maneje correctamente las relaciones bidireccionales.

Consumiendo un servicio REST

Para consumir un servicio rest desde java podemos usar la biblioteca RestTemplate de Spring, se encuentra dentro del artefacto spring-boot-starter-web

Debemos crear un Bean dentro de la configuración que nos devuelva RestTemplate

```
@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Usar RestTemplate para hacer una solicitud GET a un endpoint REST:

```
@Service
public class ApiService {

    @Autowired
    private RestTemplate restTemplate;

    public String getDataFromApi() {
        String url = "http://example.com/api/data";
        return restTemplate.getForObject(url, String.class);
    }
}
```

En el ejemplo anterior, se devuelve un String, si tuviéramos que deserializar un JSON deberíamos seguir el procedimiento visto anteriormente (creación de modelo, etc…)

CORS

CORS (Cross-Origin Resource Sharing) es una característica de seguridad implementada por los navegadores web para evitar que las páginas web realicen peticiones a un dominio diferente del que sirvió la página web. Permite a los servidores especificar quién puede acceder a sus recursos y cómo se puede acceder a ellos.

Podemos configurar CORS dentro de Springboot de varias formas:

- Individual: en cada endpoint.
- De manera global

Individual:

```
    @CrossOrigin(origins = "http://localhost:5000") // CORS
    @GetMapping("/api/tenistas")
    public ResponseEntity<?> devolverTodos(){
        return ResponseEntity.ok(tenistaService.devolverTodo());
    }
```

Global:

```
@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost:5000") //
Change this to the allowed origin
                        .allowedMethods("GET", "POST", "PUT",
"DELETE", "OPTIONS")
                        .allowedHeaders("*")
                        .allowCredentials(true);
            }
        };
    }
}
```

Validación de entrada de datos.

La funcionalidad de validación de entrada de datos en un servicio REST en Spring Boot depende del jakarta.validation en versiones más recientes. Esta funcionalidad se utiliza junto con la anotación @Valid para validar automáticamente los campos de una entidad cuando se recibe un JSON en un controlador.

Paquete de Dependencia

Para usar la validación en Spring Boot, necesitas incluir la dependencia de spring-boot-starter-validation en tu archivo pom.xml

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Anotaciones Principales de Validación

- @NotNull: Asegura que el campo no sea null.
- @NotEmpty: Asegura que el campo no sea null y que su tamaño sea mayor que cero (aplicable a colecciones, mapas, cadenas, etc.).
- @NotBlank: Asegura que el campo no sea null y que contenga al menos un carácter no blanco (aplicable solo a cadenas).
- @Size: Asegura que el tamaño de la colección, mapa, array o cadena esté dentro de los límites especificados.
- @Min: Asegura que el valor numérico sea mayor o igual al valor especificado.
- @Max: Asegura que el valor numérico sea menor o igual al valor especificado.
- @Pattern: Asegura que el valor de la cadena coincida con la expresión regular especificada.
- @Email: Asegura que el valor de la cadena sea una dirección de correo electrónico válida.

Ejemplo de uso:

```
public class UserDTO {

    @NotNull
    @Size(min = 1, max = 50)
    private String username;

    @NotBlank
    @Size(min = 8, max = 100)
    private String password;

    @NotBlank
    private String passwordConfirm;

    @Email
    @NotNull
    private String email;
}
```

```
@PostMapping("/register")
    public void registerUser(@Valid @RequestBody UserDTO userDTO) {
        try {
            userEntityService.nuevoUsuario(userDTO);
        } catch (ResponseStatusException e) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
e.getReason());
        }
    }
```

Si los datos de UserDTO no cumplen con las restricciones de validación, se lanzará una excepción y se devolverá un error de respuesta HTTP 400 (Bad Request).

Upload de archivos.

Para subir archivos a través de un servicio REST en Spring Boot, puedes seguir estos pasos:

- Configurar el controlador para manejar la subida de archivos.
- Guardar los archivos en el servidor.
- Validar el archivo subido (tamaño, extensión, MIME type).

Configurar el controlador para manejar la subida de archivos

```
@PostMapping("/upload")
    public ResponseEntity<String> uploadFile(@RequestParam("file")
MultipartFile file) {
        // Validar el archivo
        if (file.isEmpty()) {
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El archivo está
vacío");
        }

        // Validar tamaño del archivo (ejemplo: máximo 5MB)
        if (file.getSize() > 5 * 1024 * 1024) {
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El archivo es
demasiado grande");
        }

        // Validar extensión del archivo
        String fileName = file.getOriginalFilename();
        if (fileName != null && !fileName.endsWith(".txt")) {
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Solo se permiten
archivos .txt");
        }

        // Guardar el archivo en el servidor
        try {
            File dest = new File("uploads/" + fileName);
            file.transferTo(dest);
            return ResponseEntity.status(HttpStatus.OK).body("Archivo
subido exitosamente");
```

```
        } catch (IOException e) {
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error al
guardar el archivo");
        }
    }     }
```

También podemos validar el MimeType del fichero, ejemplo:

```
// Validar MIME type del archivo
        String mimeType = file.getContentType();
        if (mimeType == null || !mimeType.equals("image/jpeg")) {
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("El tipo MIME no es
válido");
        }
```

Subida de archivos y datos en el mismo form:

La anotación @RequestPart se utiliza en Spring para vincular una parte de una solicitud multipart/form-data a un parámetro del método del controlador. Es útil para manejar datos mixtos, como JSON y archivos, en una sola solicitud.

```
@PostMapping("/register")
public ResponseEntity<String> registerUser(
        @RequestPart("user") @Valid UserDTO userDTO,
        @RequestPart("profileImage") MultipartFile profileImage) {
    // Validar y guardar el archivo de imagen de perfil
    if (profileImage.isEmpty() ||
!profileImage.getContentType().equals("image/jpeg")) {
        return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Imagen de perfil
no válida");
    }

    try {
        File dest = new File("c:/uploads/" +
profileImage.getOriginalFilename());
        profileImage.transferTo(dest);
    } catch (IOException e) {
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error al
guardar la imagen de perfil");
    }

    // Guardar el usuario
    try {
        userEntityService.nuevoUsuario(userDTO);
        return ResponseEntity.status(HttpStatus.OK).body("Usuario
registrado exitosamente");
    } catch (ResponseStatusException e) {
        return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getReason());
    }
}
```