07 - Validación

Uno de los aspectos más importantes es la validación de datos. Tenemos que tener claro qué validamos o dónde validamos los datos de entrada, aunque hay veces que las respuestas no son tan sencillas como podríamos pensar.

Lo primero, ¿dónde validamos los datos? Podríamos pensar que cuánto antes mejor, y, en parte, es correcto ¿Significa eso que tenemos que validar los datos en los controladores? ¿Qué pasa si cambiamos los controladores o si tenemos varios controladores que reciben los mismos datos? Eso nos obligaría a repetir el código de validación en varios controladores, con lo que no parece la mejor opción.

Lo siguiente que podemos valorar es hacer las validaciones en los servicios. Y sí, tendremos algunas validaciones en nuestros servicios, pero pensemos en los campos *birhtYear* y *deathYear* de directores y actores. Nuestra validación básica debería comprobar que ambos campos reciben años válidos y que *deathYear* no es anterior a *birthYear*. En realidad, ninguna de esas validaciones afectan al servicio que vaya a tratar con esos datos. La primera afecta sólo a los atributos concretos, mientras que la segunda afectaría a dos atributos de nuestra entidad.

Entonces, ¿dónde hacemos las validaciones? La respuesta es sencilla: en varios sitios (atributos, entidades, servicios...) Ésto nos plantea un problema. Si tenemos validaciones en diferentes clases, después puede ser complicado encontrar esas validaciones si queremos modificarlas, borrarlas...

Para intentar simplificar el problema anterior, seguiremos una regla básica que nos facilite encontrar dichas validaciones: **Las validaciones siempre las pondremos en el nivel más bajo posible**. Es decir, intentaremos que las validaciones sean siempre en los atributos. Si no podemos, elevaremos esa validación a los modelos de datos. Por último, si una validación afecta a varias entidades y no podemos ponerla en una concreta, lo haremos en los servicios.

Jakarta Validation

Empecemos por las validaciones más básicas, como las de los años. Tenemos claro que esas validaciones deberían estar en los atributos, pero, ¿cómo las hacemos? Para ayudarnos en la tarea, tenemos Jakarta Bean Validation (https://beanvalidation.org/).

Jakarta Bean Validation es una especificación de Java que define un conjunto de APIs y anotaciones para la validación de objetos en aplicaciones Java. Proporciona un marco estándar para la validación de datos, permitiendo a los desarrolladores especificar reglas de validación directamente en las clases de dominio de sus aplicaciones.

La especificación *Jakarta Bean Validation* se centra en la validación de datos de manera declarativa mediante el uso de anotaciones en los campos de las clases. Algunas de las anotaciones comunes incluyen **@NotNull** para garantizar que un valor no sea nulo, **@Size** para especificar restricciones sobre la longitud de una cadena, **@Min** y **@Max** para establecer límites numéricos, entre otras.

Diferentes implementaciones, como Hibernate Validator (https://hibernate.org/validator/), Apache BVal (https://bval.apache.org/), y Apache Geronimo Validator (https://geronimo.apache.org/apidocs/2.0.1/org/apache/geronimo/validator/package-summary.html), proporcionan el soporte concreto para la especificación *Jakarta Bean Validation*, permitiendo a los desarrolladores elegir la implementación que mejor se adapte a sus necesidades.

En nuestro caso, vamos a utlizar *spring-boot-starter-validation*, que está basado en *Hibernate Validator*. Lo primero, será añadir la dependencia correspondiente:

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

Ahora ya podemos añadir anotaciones en nuestros atributos para validarlos. Por ejemplo, si queremos validar que los años de nuestras entidades puedan ser nulos y deban ser posteriores a 1880:

```
import jakarta.annotation.Nullable;
import jakarta.validation.constraints.Min;

public class Actor {

    Integer id;
    private String name;

    @Nullable
    @Min(value = 1880, message = "El año debe ser posterior a 1
    private Integer birthYear;

    @Nullable
    @Min(value = 1880, message = "El año debe ser posterior a 1
    private Integer deathYear;
```

Aquí (https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraints/package-summary) puedes ver un listado de las diferentes etiquetas que puedas utilizar.

¿Y qué pasa si queremos hacer validaciones más complicadas? Para esos casos, tenemos la posibilidad de crear nuestras propias etiquetas de validación. Lo primero que tenemos que hacer es crear una clase que implemente ConstraintValidator (https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraintvalidator):

```
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class YearValidator implements ConstraintValidator<ValidY
```

Ésto nos obligará a implementar dos métodos: **initialize()** y **isValid()**:

```
public class YearValidator implements ConstraintValidator<Valid
    @Override
    public void initialize(ValidYear constraintAnnotation) {
        // Método de inicialización, puedes implementar lógica
    }

    @Override
    public boolean isValid(Integer year, ConstraintValidatorCon
        // Método de validación, implementa la lógica de valida
        // Por ejemplo, verifica si el año es mayor o igual a 1
        return year != null && year >= 1900;
    }
}
```

En la implementación del método *initialize()*, es una buena práctica llamar al método *initialize()* de la clase base ConstraintValidator utilizando ConstraintValidator.super.initialize(constraintAnnotation);. Esto asegura que cualquier lógica de inicialización definida en las clases base se ejecute correctamente.

El método crucial es *isValid()*, que contiene la lógica de validación personalizada. Por ejemplo, si queremos verificar que un año válido sea null o posterior a 1880, nuestro validador quedará:

```
public class YearValidator implements ConstraintValidator<Valid

    @Override
    public void initialize(ValidYear constraintAnnotation) {
        ConstraintValidator.super.initialize(constraintAnnotati
    }

    @Override
    public boolean isValid(Integer year, ConstraintValidatorCon
        return (year == null || (year >= 1850 && year <= 9999))
    }
}
```

El siguiente paso será el código para crear la anotación:

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = YearValidator.class)
public @interface ValidYear {
    String message() default "El año debe ser posterior a 1850"

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Algunos puntos clave sobre la anotación que acabamos de crear:

- **@Target({ElementType.METHOD, ElementType.FIELD})**: Indica que la anotación se puede aplicar a métodos y campos.
- **@Retention(RetentionPolicy.RUNTIME)**: Indica que la anotación estará disponible en tiempo de ejecución.
- **@Constraint(validatedBy = YearValidator.class)**: Especifica que la clase *YearValidator* es la que proporciona la lógica de validación para esta anotación.
- **message() default "El año debe ser posterior a 1850"**: Proporciona un mensaje predeterminado que se mostrará si la validación falla.
- **groups() default {} y payload() default {}**: Son atributos estándar de anotaciones de validación en Jakarta Validation.

Ahora podemos usar **@ValidYear** para marcar campos en nuestras clases y la validación personalizada definida en *YearValidator* se aplicará según la lógica que hayamos implementado:

```
1  public class Director {
2
3      @Nullable
4      Integer id;
5
6      private String name;
7
8      @ValidYear
9      private Integer birthYear;
10
11     @ValidYear
12     private Integer deathYear;
```

> Si estamos utilizando DTOs, estas validaciones las podemos poner en esas clases, en lugar de en las entidades del dominio. Otra opción común es utilizar Value Objects (https://en.wikipedia.org/wiki/Value_object), concepto fundamental en el diseño orientado a dominio (DDD). Representan objetos cuya identidad está determinada por sus atributos, y no por su identificador único. En ese caso, las validaciones las podríamos hacer directamente en los *value objects*.

Validaciones a nivel entidad

En el punto anterior vimos como validar atributos individuales. Vamos a ver ahora como podemos comprobar reglas que afectan a varios atributos. Por ejemplo, el año de fallecimiento de un director no puede ser anterior al año de nacimiento. Para eso, podemos validar esa condición en los *setters* correspondientes:

```java
@Data
public class Director {

    @Nullable
    Integer id;
    private String name;

    @ValidYear
    private Integer birthYear;

    @ValidYear
    private Integer deathYear;

    public void setBirthYear(Integer birthYear) {
        if(this.deathYear != null && birthYear!= null && this.d
            throw new ValidationException("El año de nacimiento
        }
        this.birthYear = birthYear;
    }

    public void setDeathYear(Integer deathYear) {
        if(this.birthYear != null && deathYear != null &&  this
            throw new ValidationException("El año de nacimiento
        }
        this.deathYear = deathYear;
    }
}
```

Básicamente, comprobamos en ambos *setters* si el año de nacimiento es posterior al año de fallecimiento (o si es null, en ambos casos). Si se cumple, lanzamos una excepción.

¿Qué pasa si una validación depende de varios atributos de diferentes entidades? Depende. Por ejemplo, queremos comprobar que el año de nacimiento de un director no puede ser posterior al año de estreno de una película cuando se le agrega éste (el director). En ese caso, podemos hacer la comprobación en la misma entidad, ya que en el método *setDirector()* de la entidad *movie* tenemos todos los datos:

```java
public class Movie {

...

     public void setDirector(Director director) {
        if(director.getBirthYear() != null && director.getBirth
            throw new ValidationException("El año de nacimiento
        }
        this.director = director;
    }

...
```

Validaciones a nivel servicio

En nuestro caso, una misma persona puede estar en la tabla director y actor. Queremos comprobar que, al añadir un nuevo registro a alguna de esas tablas, los datos sean iguales al de la otra tabla (en caso de existir). En ese caso, no podemos

hacer la validación ni en los atributos ni en las entidades. Tenemos que hacer la validación en el servicio correspondiente:

```
@Override
public Actor create(Actor actor) {
    Director director = directorRepository.findByName(actor.get
    if(director != null) {
        if (
                director.getBirthYear() != actor.getBirthYear()
                        director.getDeathYear() != actor.getDea

        ) {
            throw new ValidationException("Datos incorrectos");
        }
    }
    ...
```

En el ejemplo anterior, estamos comprobando si existe el actor en la tabla directores. Si es así, comprobamos que las fechas de nacimiento y fallecimiento sean las mismas.

> Obviamente, el ejemplo anterior podría fallar si existen dos personas con el mismo nombre y apellidos, pero es sólo eso, un ejemplo de cómo hacer validaciones a nivel servicio.

clase/daw/dws/2eval/validacion.txt Última modificación: 2025/01/11 10:51 por cesguiro

cesguiro