(/)

Interpolación de mensajes de validación de primavera

Última modificación: 28 de septiembre de 2019

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Primavera (https://www.baeldung.com/category/spring/) +

1. Introducción

La interpolación de mensajes es el proceso utilizado para crear mensajes de error para las (https://www.baeldung.com/javax-validation) restricciones de validación de beans Java (https://www.baeldung.com/javax-validation) . Por ejemplo, podemos ver los mensajes proporcionando un valor *nulo* para un campo anotado con la anotación *javax.validation.constraints.NotNull* .

En este tutorial, aprenderemos cómo usar la interpolación de mensajes Spring predeterminada y cómo crear nuestro propio mecanismo de interpolación.

Para ver ejemplos de otras bibliotecas que proporcionan restricciones además de *javax.validation* , eche un vistazo a Restricciones específicas del validador de Hibernate (https://www.baeldung.com/hibernate-validator-constraints) . También podemos crear una anotación personalizada de validación de Spring (https://www.baeldung.com/spring-mvc-custom-validator) .

2. Interpolación de mensajes predeterminada

Antes de entrar en fragmentos de código, consideremos un ejemplo de una respuesta HTTP 400 con un *mensaje* predeterminado de violación de restricción *@NotNull* :

```
{
    ....
    "status": 400,
    "error": "Bad Request",
    "errors": [
        {
            ....
            "defaultMessage": "must not be null",
            ....
        }
    ],
    "message": "Validation failed for object='notNullRequest'. Error count: 1",
    ....
}
```

Spring recupera los detalles del mensaje de violación de restricción de los descriptores del mensaje. Cada restricción define su descriptor de mensaje predeterminado utilizando el atributo de *mensaje* . Pero, por supuesto, podemos sobrescribirlo con un valor personalizado.

Como ejemplo, crearemos un controlador REST simple con un método POST:

```
@RestController
public class RestExample {

    @PostMapping("/test-not-null")
    public void testNotNull(@Valid @RequestBody NotNullRequest request) {
        // ...
    }
}
```

El cuerpo de la solicitud se asignará al objeto *NotNullRequest* , que solo tiene una *Cadena* archivada anotada con *@NotNull* :

```
public class NotNullRequest {

    @NotNull(message = "stringValue has to be present")
    private String stringValue;

    // getters, setters
}
```

Ahora, cuando enviamos una solicitud POST que no pasa esta verificación de validación, veremos nuestro mensaje de error personalizado:

```
{
    ...
    "errors": [
        {
            ...
            "defaultMessage": "stringValue has to be present",
            ...
        }
    ],
    ...
}
```

El único valor que cambia es *defaultMessage* . Pero aún obtenemos mucha información sobre códigos de error, nombre de objeto, nombre de campo, etc. Para limitar el número de valores mostrados, podemos implementar el Manejo de mensajes de error personalizados para la API REST (https://www.baeldung.com/global-error-handler-in-a-spring-rest-api) .

3. Interpolación con expresiones de mensaje

En Spring, podemos **usar el lenguaje de expresión unificada para definir nuestros descriptores de mensajes** . Esto permite definir **mensajes de error basados en lógica condicional y también permite opciones de formato avanzadas** .

Para entenderlo más claramente, veamos algunos ejemplos.

En cada anotación de restricción, podemos acceder al valor real de un campo que se está validando:

```
@Size(
  min = 5,
  max = 14,
  message = "The author email '${validatedValue}' must be between {min} and {max} characters long"
)
private String authorEmail;
```

Nuestro mensaje de error contendrá tanto el valor real de la propiedad como los parámetros *mínimo* y *máximo* de la anotación *@Size* :

```
"defaultMessage": "The author email 'toolongemail@baeldung.com' must be between 5 and 14 characters
```

Observe que para acceder a variables externas, usamos la sintaxis *$ {}* , pero para acceder a otras propiedades desde la anotación de validación, usamos *{}* .

El uso del operador ternario también es posible:

```
@Min(
  value = 1,
  message = "There must be at least {value} test{value > 1 ? 's' : ''} in the test case"
)
private int testCount;
```

Spring convertirá el operador ternario en un solo valor en el mensaje de error:

```
"defaultMessage": "There must be at least 2 tests in the test case"
```

También podemos llamar métodos sobre variables externas:

```
@DecimalMin(
  value = "50",
  message = "The code coverage ${formatter.format('%1$.2f', validatedValue)} must be higher than {val
)
private double codeCoverage;
```

La entrada no válida generará un mensaje de error con el valor formateado:

```
"defaultMessage": "The code coverage 44.44 must be higher than 50%"
```

Como podemos ver en estos ejemplos, algunos caracteres como *{,}, $* y */* se usan en expresiones de mensaje, por lo que debemos escapar de ellos con un carácter de barra diagonal inversa antes de usarlos literalmente: *\ {, \}, \ $,* y *\ * .

4. Interpolación de mensajes personalizados

En algunos casos, queremos **implementar un motor de interpolación de mensajes personalizado** . Para hacerlo, primero debemos implementar la interfaz *javax.validation.MessageInterpolation* :

```
public class MyMessageInterpolator implements MessageInterpolator {
    private final MessageInterpolator defaultInterpolator;

    public MyMessageInterpolator(MessageInterpolator interpolator) {
        this.defaultInterpolator = interpolator;
    }

    @Override
    public String interpolate(String messageTemplate, Context context) {
        messageTemplate = messageTemplate.toUpperCase();
        return defaultInterpolator.interpolate(messageTemplate, context);
    }

    @Override
    public String interpolate(String messageTemplate, Context context, Locale locale) {
        messageTemplate = messageTemplate.toUpperCase();
        return defaultInterpolator.interpolate(messageTemplate, context, locale);
    }
}
```

En esta implementación simple, solo estamos cambiando el mensaje de error a mayúsculas. Al hacerlo, nuestro mensaje de error se verá así:

```
"defaultMessage": "THE CODE COVERAGE 44.44 MUST BE HIGHER THAN 50%"
```

También necesitamos **registrar nuestro interpolador** en la fábrica de *validación javax.validation* :

```
Validation.byDefaultProvider().configure().messageInterpolator(
  new MyMessageInterpolator(
    Validation.byDefaultProvider().configure().getDefaultMessageInterpolator())
);
```

5. Conclusión

En este artículo, hemos aprendido cómo funciona la interpolación de mensajes Spring predeterminada y cómo crear un motor de interpolación de mensajes personalizado.

Y, como siempre, todo el código fuente está disponible en GitHub (https://github.com/eugenp/tutorials/tree/master/spring-mvc-simple-2) .

¡Los comentarios están cerrados en este artículo!

KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)