Arquitecturas de Software para Aplicaciones Empresariales

Validaciones con Bean Validator

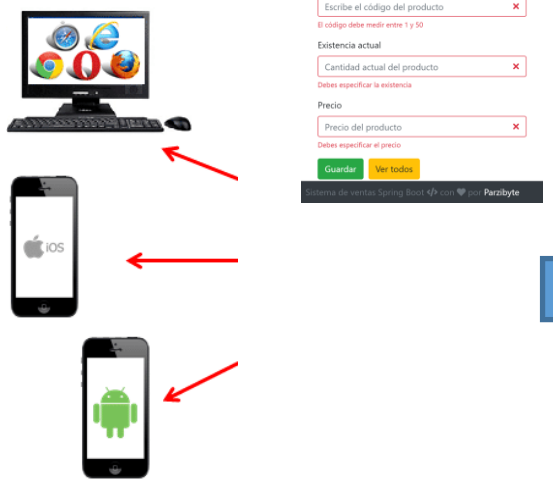
PROGRAMA
DE INGENIERIA DE SISTEMAS

Ing. Daniel Eduardo Paz Perafán (danielp@Unicauca.edu.co)

Ing. Pablo A. Magé (pmage@Unicauca.edu.co)

Donde realizar las validaciones

FRONTEND



BACKEND

BASE DE DATOS

¿Cómo modificamos la respuesta HTTP?

Introducción

- Bean Validation es el estándar de facto para implementar la lógica de validación en el ecosistema Java

- JSR 380 es una especificación de la API de Java para la validación de beans, que forma parte de Jakarta EE y JavaSE. Esto asegura que las propiedades de un bean cumplen con criterios específicos, utilizando anotaciones como @NotNull , @Min y @Max .

- Esta versión requiere Java 8 o superior y aprovecha las nuevas funciones agregadas en Java 8, como las anotaciones de tipo y la compatibilidad con nuevos tipos como Optional y LocalDate .

Librerias

Según la especificación JSR 380, la dependencia validation-api contiene las API de validación estándar:

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>2.0.1.Final</version>
 </dependency>
```

Hibernate Validator es la implementación de referencia de la API de validación. Para usarlo, necesitamos agregar la siguiente dependencia:

```
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>6.0.13.Final</version>
 </dependency>
```

hibernate-validator está completamente separado de los aspectos de persistencia de Hibernate. Entonces, al agregarlo como una dependencia, no agregamos estos aspectos de persistencia al proyecto.

Anotaciones para validar atributos

La especificación Bean Validation define un conjunto de anotaciones que proporcionan una colección de restricciones genéricas y básicas y que podemos comprobar en la documentación.

```java
@Entity
@Table(name = "Usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @NotNull
    @Size(min = 5, max = 45)
    @Column(nullable = false, length = 45)
    private String username;
    @NotNull
    @Email
    private String email;
    @PastOrPresent
    private Date fechaRegistro;
    @Pattern(regexp = "[6][0-9]{8}")
    private String telefono;
    @PositiveOrZero
    private float salario;
```

```java
import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.PastOrPresent;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.PositiveOrZero;
import javax.validation.constraints.Size;
```

Anotaciones para validar atributos

La validación la debemos colocar en las clases DTO, hay que evitar colocarlas en las clases Entity.

```java
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ClienteDTO {
    private Integer id;
    @NotNull(message = "{user.name.empty}")
    @Size(min = 5, max = 45, message = "la cantidad de caracteres del nombre debe estar entre 5 y 45")
    private String nombre;
    @NotNull(message = "{user.lastname.empty}")
    @Size(min = 5, max = 45, message = "{user.lastname.length}")
    private String apellido;
    @NotNull(message = "{user.email.emply}")
    @Email(message = "{user.email.mask}")
    private String email;
    @PastOrPresent(message = "{user.date.past}")
    private Date createAt;
    @Pattern(message = "{user.telephone.pattern}", regexp = "[6][0-9]{8}")
    private String telefono;
    @PositiveOrZero(message = "{user.salary.positive}")
    private float salario;
```

Anotaciones para validar atributos


```java
@NotNull
@Size(min = 5, max = 45)
@Column(nullable = false, length = 45)
private String username;
```

El nombre es un campo obligatorio con un tamaño comprendido entre 5 y 45 caracteres. Si simplemente queremos que contenga al menos un carácter podemos utilizar @NotBlank

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

El email es obligatorio y debe tener el correspondiente formato.

```java
@PastOrPresent
private Date fechaRegistro;
```

La fecha de registro debe estár en el pasado o en el presente

```java
@Pattern(regexp = "[6][0-9]{8}")
private String telefono;
```

El número de teléfono es opcional pero si se proporciona debe ser un número de 9 dígitos que empiece por seis.

```java
@PositiveOrZero
private float salario;
```

El salario debe ser un número 0 o un número positivo

Anotaciones para validar atributos

Todas las anotaciones utilizadas en el ejemplo son anotaciones JSR estándar:

@NotNull valida que el valor de la propiedad anotado no es *nulo* .
@NotEmpty valida que la propiedad no sea nula ni esté vacía; se puede aplicar avalores de *cadena* , *colección* , *mapa* o *matriz* .
@NotBlank se puede aplicar solo a valores de texto y valida que la propiedad no sea nula ni carente de espacios en blanco.
@AssertTrue valida que el valor de la propiedad anotado sea *verdadero.*
@Size valida que el valor de la propiedad anotado tiene un tamaño entre los atributos *min* y *max* ; se puede aplicar a las propiedades de *cadena* , *colección* , *mapa* y matriz.
@Min valida que la propiedad anotada tiene un valor no menor que el atributo *value* .
@Max valida que la propiedad anotada tiene un valor no mayor que el atributo *value* .
@Email valida que la propiedad anotada es una dirección de correo electrónico válida.

Algunas anotaciones aceptan atributos adicionales, pero el atributo de *mensaje* es común a todos ellos. Este es el mensaje que normalmente se presentará cuando el valor de la propiedad respectiva falle en la validación.

Anotaciones para validar atributos

Algunas anotaciones adicionales que se pueden encontrar en JSR:

@Pattern plantea que un campo de cadena solo es válido cuando coincide con una determinada expresión regular.
@Positive y @PositiveOrZero se aplican a valores numéricos y validan que sean estrictamente positivos, o positivos incluyendo 0.
@Negative y @NegativeOrZero se aplican a valores numéricos y validan que sean estrictamente negativos, o negativos incluyendo 0.
@Past y @PastOrPresent validan que un valor de fecha está en el futuro, o en el futuro, incluido el presente.
@Future y @FutureOrPresent validan que un valor de fecha está en el pasado o en el pasado, incluido el presente.

Anotaciones @Validated y @Valid

- Podemos informarle a Spring que queremos validar un determinado objeto mediante las anotaciones @Validated y @Valid.

- La @Validated es una anotación a nivel de clase que podemos usar para decirle a Spring que valide los parámetros que se pasan a un método de la clase anotada.

- La @Valid anotación en los parámetros y campos del método podemos usarla para decirle a Spring que queremos que se valide un parámetro del método.

Validación del cuerpo de la petición

Uso de la anotación @Valid para validar el cuerpo de la solicitud

Para validar el cuerpo de la solicitud de una solicitud HTTP entrante, anotamos el cuerpo de la solicitud con la @Valid anotación en un controlador REST.

```
@PostMapping("/usuarios")
public ResponseEntity<?> create(@Valid @RequestBody Usuario cliente) {
    Usuario objUsuario = null;
    HashMap<String, Object> respuestas= new HashMap();
    ResponseEntity<?> objRespuesta;
    try
    {
        objUsuario =  UsuarioService.save(cliente);
        objRespuesta= new ResponseEntity<Usuario>(objUsuario,HttpStatus.CREATED);
    }
    catch(DataAccessException e)
    {
        respuestas.put("mensaje", "Error al realizar la inserción en la base de datos");
        respuestas.put("descripción del error", e.getMessage());
        objRespuesta= new ResponseEntity<HashMap<String, Object>>(respuestas,HttpStatus.BAD_REQUEST);

    }

    return objRespuesta;
}
```

Validación del cuerpo de la petición

- En Spring MVC, tenemos muchas formas de establecer el código de estado de una respuesta HTTP. Una forma es usando la anotación @ResponseStatus.

- Para capturar excepciones utilizamos una solución a nivel @Controller. Definiremos un método para manejar excepciones y anotarlo con @ExceptionHandler:

- Solo está activo para ese controlador en particular, no globalmente para toda la aplicación. Por supuesto, agregar esto a cada controlador hace que no sea adecuado para un mecanismo general de manejo de excepciones.

Validación del cuerpo de la petición

Analizando el ejemplo del servicio que permite crear un usuario, retorna un código 201 o 400

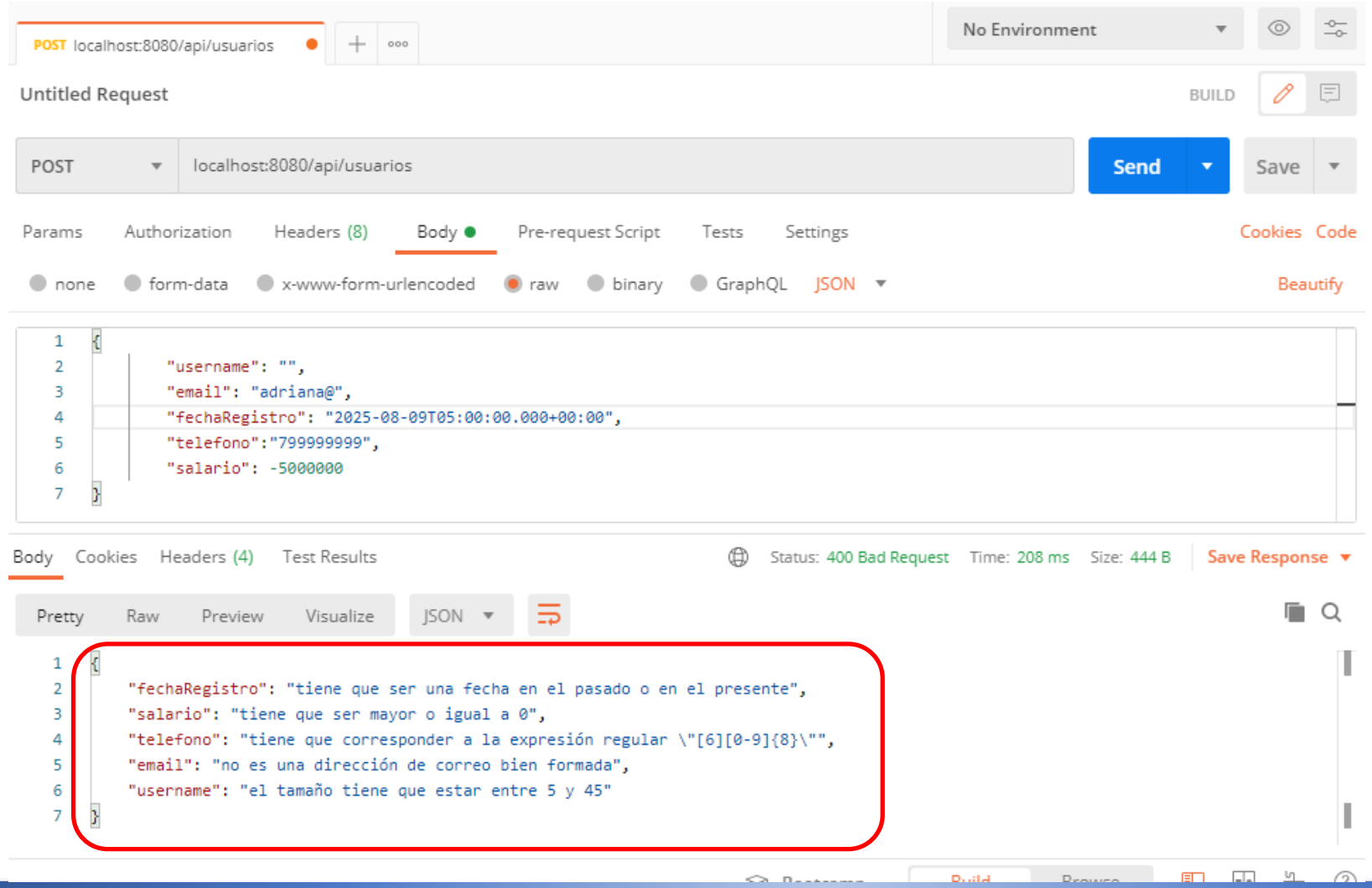
Si la validación falla, activará una excepción MethodArgumentNotValidException. De forma predeterminada, Spring traducirá esta excepción a un estado HTTP 400 (Bad Request).

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError ) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return errors;
}
```

La anotación *@ExceptionHandler* nos permite manejar tipos específicos de excepciones a través de un solo método. Por tanto, podemos utilizarlo para procesar los errores de validación:

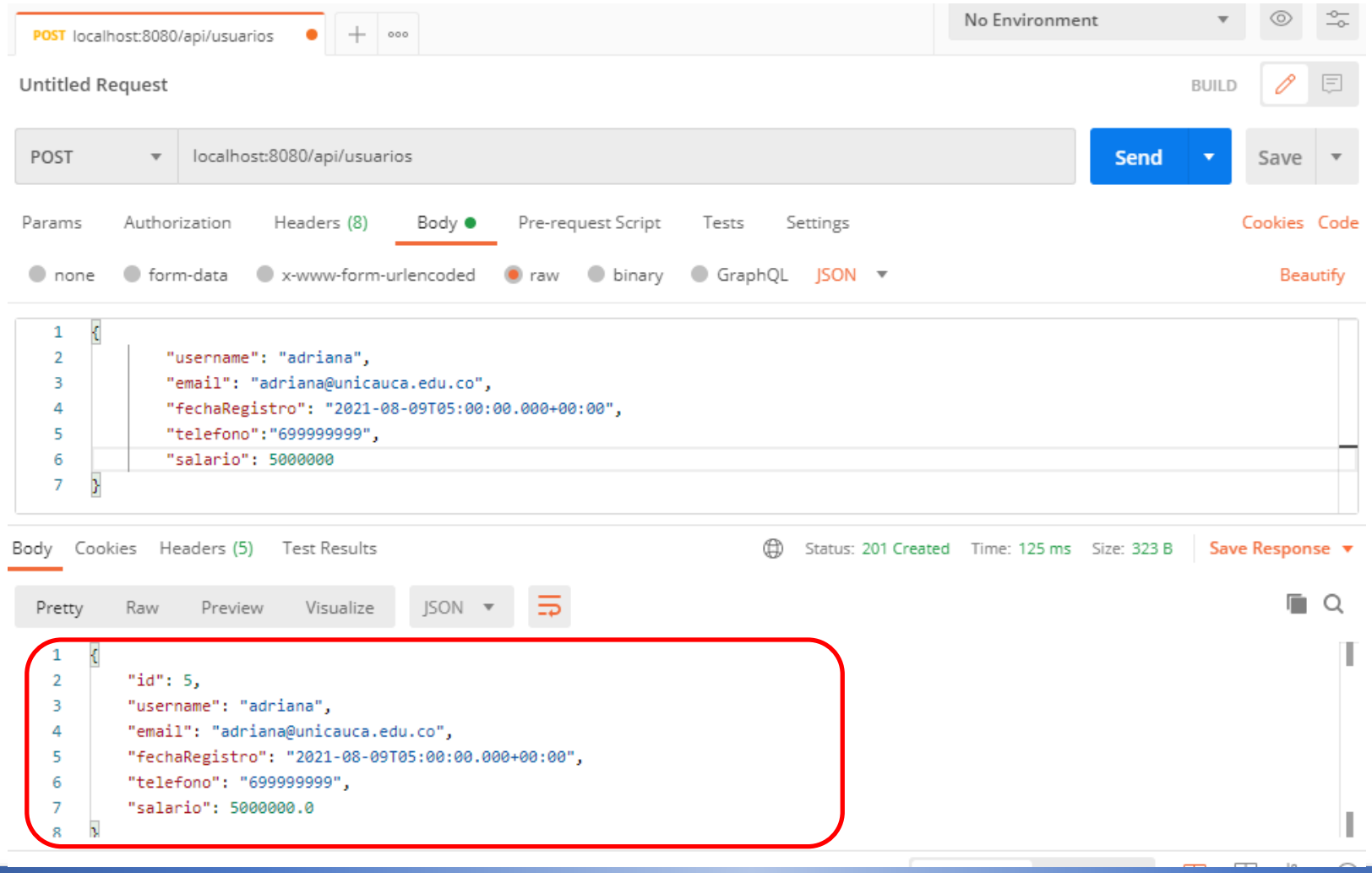
Validación del cuerpo de la petición

Prueba



Validación del cuerpo de la petición

Prueba



Validación de variables de ruta y parámetros

La validación de variables de ruta y parámetros de solicitud funciona de manera un poco diferente.

```java
@RestController
@RequestMapping("/api")
@Validated
public class UsuarioRestController {

    @Autowired
    private IUuariosServices UsuarioService;

    @GetMapping("/usuarios")
    public List<Usuario> index() {
        return UsuarioService.findAll();
    }
```

```java
    @GetMapping("/usuarios/{id}")
    public ResponseEntity<?> show(@PathVariable @Min(5) Integer id) {
        Usuario objUuario = null;
        HashMap<String, Object> respuestas= new HashMap();
        ResponseEntity<?> objRespuesta;
        try{
            objUuario = UsuarioService.findById(id);

            if(objUuario==null)
            {
                respuestas.put("mensaje", "El usuario con ID: "+id+" no existe en la base de datos");
                objRespuesta= new ResponseEntity<HashMap<String, Object>>(respuestas,HttpStatus.NOT_FOUND);
            }
            else
            {
                objRespuesta= new ResponseEntity<Usuario>(objUuario,HttpStatus.OK);
            }

        }
        catch(DataAccessException e)
        {
            respuestas.put("mensaje", "Error al realizar la consulta en la base de datos");
            respuestas.put("descripción del error", e.getMessage());
            objRespuesta= new ResponseEntity<HashMap<String, Object>>(respuestas,HttpStatus.INTERNAL_SERVER_ERROR);
        }

        return objRespuesta;
    }
```

Validación de variables de ruta y parámetros

La validación de variables de ruta y parámetros de solicitud funciona de manera un poco diferente.

No estamos validando objetos complejos de Java en este caso, ya que las variables de ruta y los parámetros de solicitud son tipos primitivos como int o sus objetos homólogos como Integer o String.

En lugar de anotar un campo de clase como el anterior, estamos agregando una anotación de restricción (en este caso @Min) directamente al parámetro del método en el controlador Spring:

Tenemos que agregar la @Validated anotación de Spring al controlador a nivel de clase para decirle a Spring que evalúe las anotaciones de restricción en los parámetros del método.

Validación de variables de ruta y parámetros

A diferencia de la validación del cuerpo de la solicitud, una validación fallida activará ConstraintViolationException.

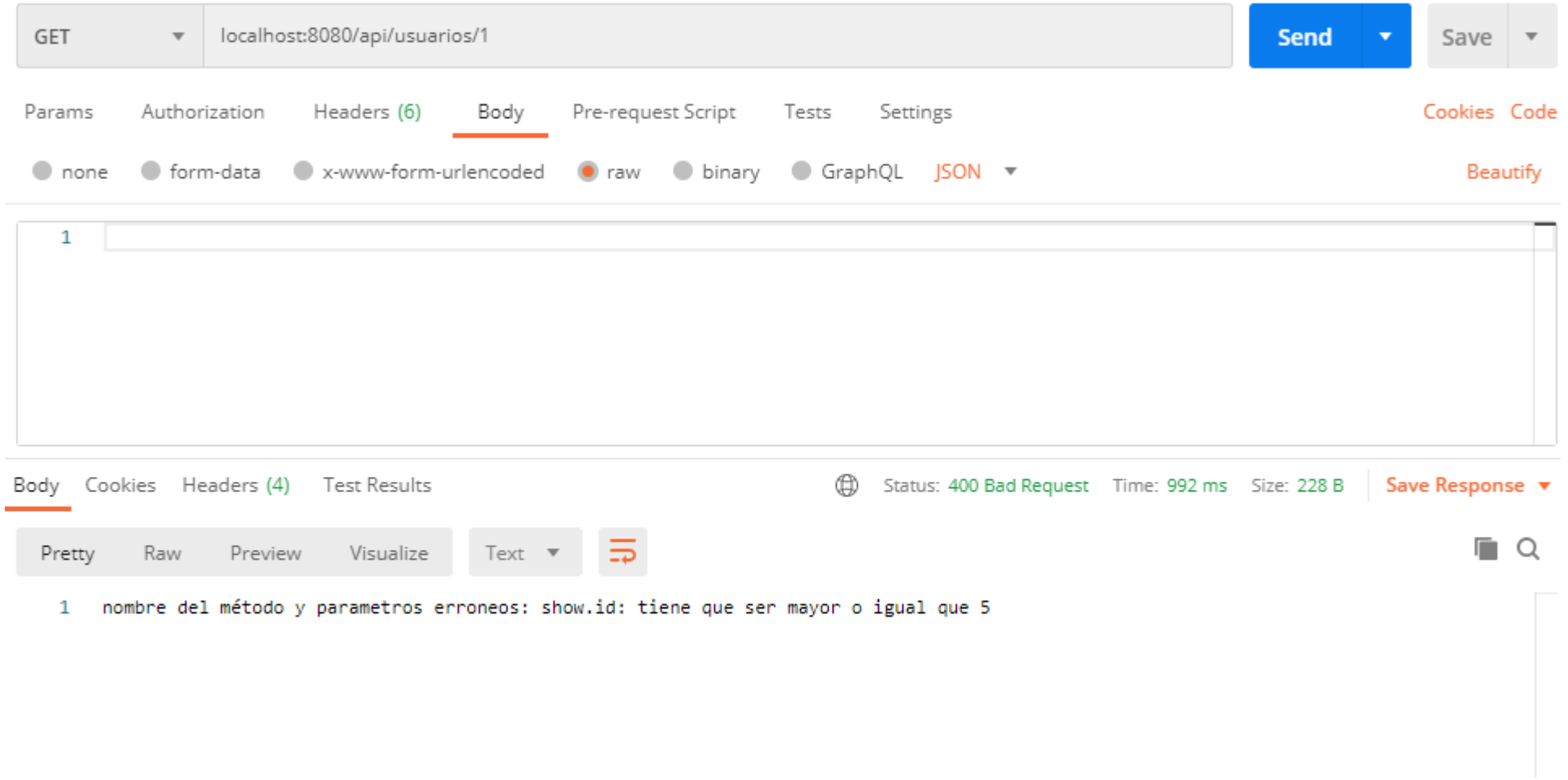
Spring no registra un controlador de excepción predeterminado para esta excepción, por lo que de manera predeterminada causará una respuesta con el estado HTTP 500 (Error interno del servidor).

Si queremos devolver un estado HTTP 400 en su lugar (lo cual tiene sentido, ya que el cliente proporcionó un parámetro no válido, lo que lo convierte en una solicitud incorrecta), podemos agregar un método de excepción personalizado a nuestro controlador:

```
@ExceptionHandler(ConstraintViolationException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
ResponseEntity<String> handleConstraintViolationException(ConstraintViolationException e) {
    return new ResponseEntity<>("nombre del método y parametros erroneos: " + e.getMessage(), HttpStatus.BAD_REQUEST);
}
```

Validación de variables de ruta y parámetros

Prueba



Agregar mensajes a las validaciones

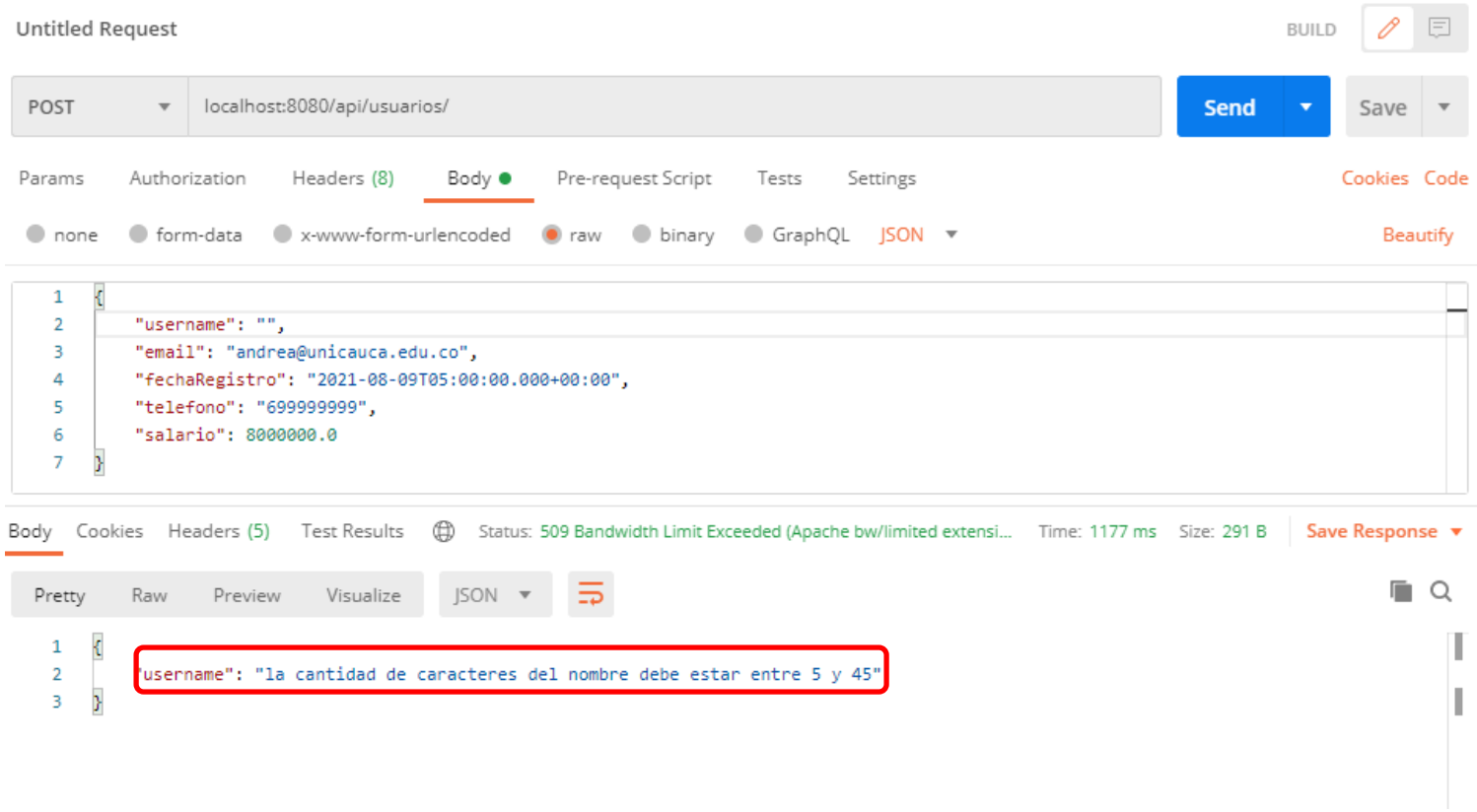
Por defecto las anotaciones proporcionan un mensaje de error genérico y descriptivo, pero generalmente vamos a querer definir nuestros propios mensajes de error personalizados. Esto podemos hacerlo utilizando el atributo *message* de las anotaciones.

```
@Entity
@Table(name = "Usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @NotNull(message = "el nombre del usuario es obligatorio")
    @Size(min = 5, max = 45, message="la cantidad de caracteres del nombre debe estar entre 5 y 45")
    @Column(nullable = false, length = 45)
    private String username;
    @NotNull
    @Email
    private String email;
    @PastOrPresent
    private Date fechaRegistro;
    @Pattern(regexp = "[6][0-9]{8}")
    private String telefono;
    @PositiveOrZero
    private float salario;
```

Agregar mensajes a las validaciones

Prueba

Nos retorna el servidor el mensaje personalizado



Agregar mensajes personalizados a las validaciones

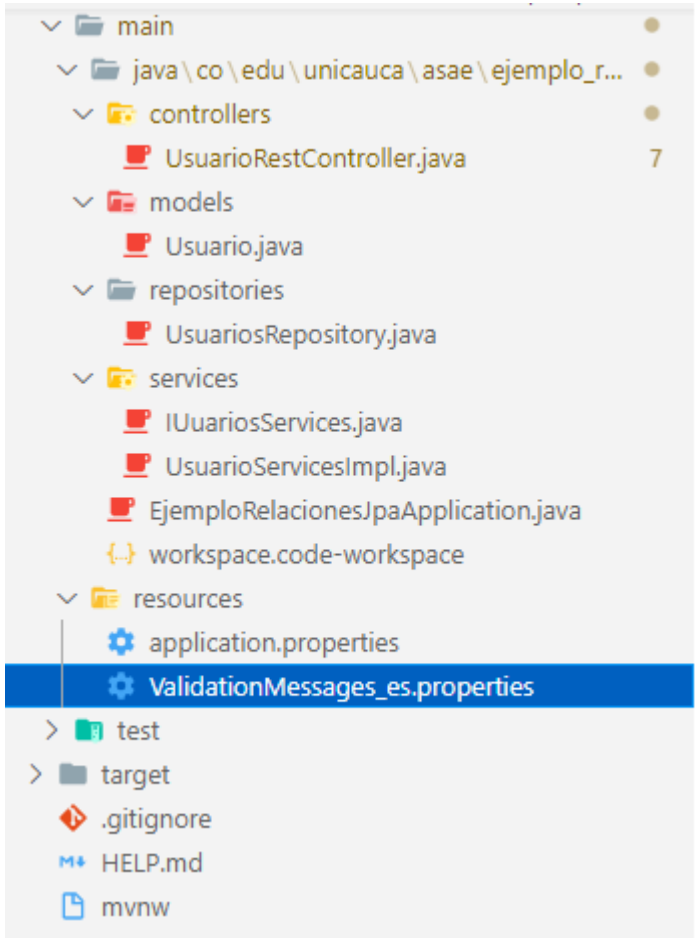
Definir el mensaje que verá el usuario tal cual es una mala práctica ya que habitualmente tendremos que localizar nuestras aplicaciones en varios idiomas.

Además, tenemos los mensajes codificados dentro del código fuente lo que obliga a recompilarlo ante cualquier cambio en los mismos.

Por ello se debe utilizar una clave que permita identificar el mensaje de error a mostrar, y obtener el texto de ese mensaje en el idioma adecuado a partir de ficheros de localización.

```
@Entity
@Table(name = "Usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @NotNull(message = "{user.name.empty}")
    @Size(min = 5, max = 45, message="la cantidad de caracteres del nombre debe estar entre 5 y 45")
    @Column(nullable = false, length = 45)
    private String username;
    @NotNull(message = "{user.email.emply}")
    @Email(message = "{user.email.mask}")
    private String email;
    @PastOrPresent(message = "{user.date.past}")
    private Date fechaRegistro;
    @Pattern(message = "{user.telephone.pattern}",regexp = "[6][0-9]{8}")
    private String telefono;
    @PositiveOrZero (message = "{user.salary.positive}")
    private float salario;
```

Agregar mensajes personalizados a las validaciones



En el classpath de la aplicación (/src/main/resources) creamos para cada idioma ficheros .properties. El fichero con los textos por defecto será ValidationMessages.properties, mientras que para cada idioma seguimos la convención de crear un fichero con el nombre «ValidationMessages» seguido del código de localización según la respuesta de la llamada a Locale.getDefault(). Las tildes van en Unicode.

ValidationMessages_es.properties

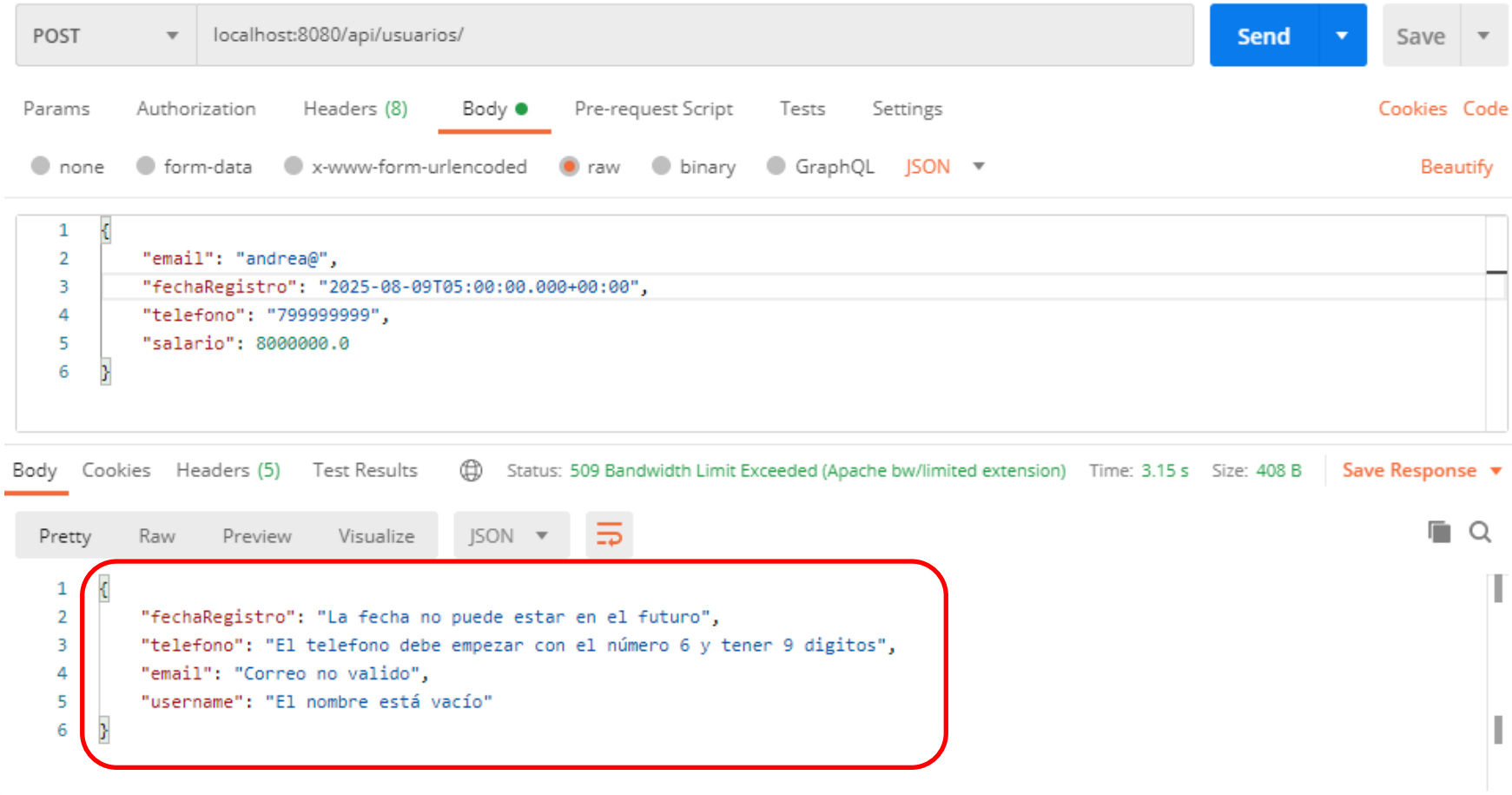
src > main > resources > ValidationMessages_es.properties

```
1  user.name.empty = El nombre est\u00E1 vac\u00EDo
2  user.email.mask= Correo no valido
3  user.email.emply= El correo est\u00E1 vac\u00EDo
4  user.date.past=La fecha no puede estar en el futuro
5  user.telephone.pattern= El telefono debe empezar con el número 6 y tener 9 digitos
6  user.salary.positive= El salario no puede ser negativo
```

Agregar mensajes personalizados a las validaciones

Prueba

Los mensajes de los errores se resuelven en función del locale que tenga configurado Java por defecto en el momento de ejecutarse la validación.

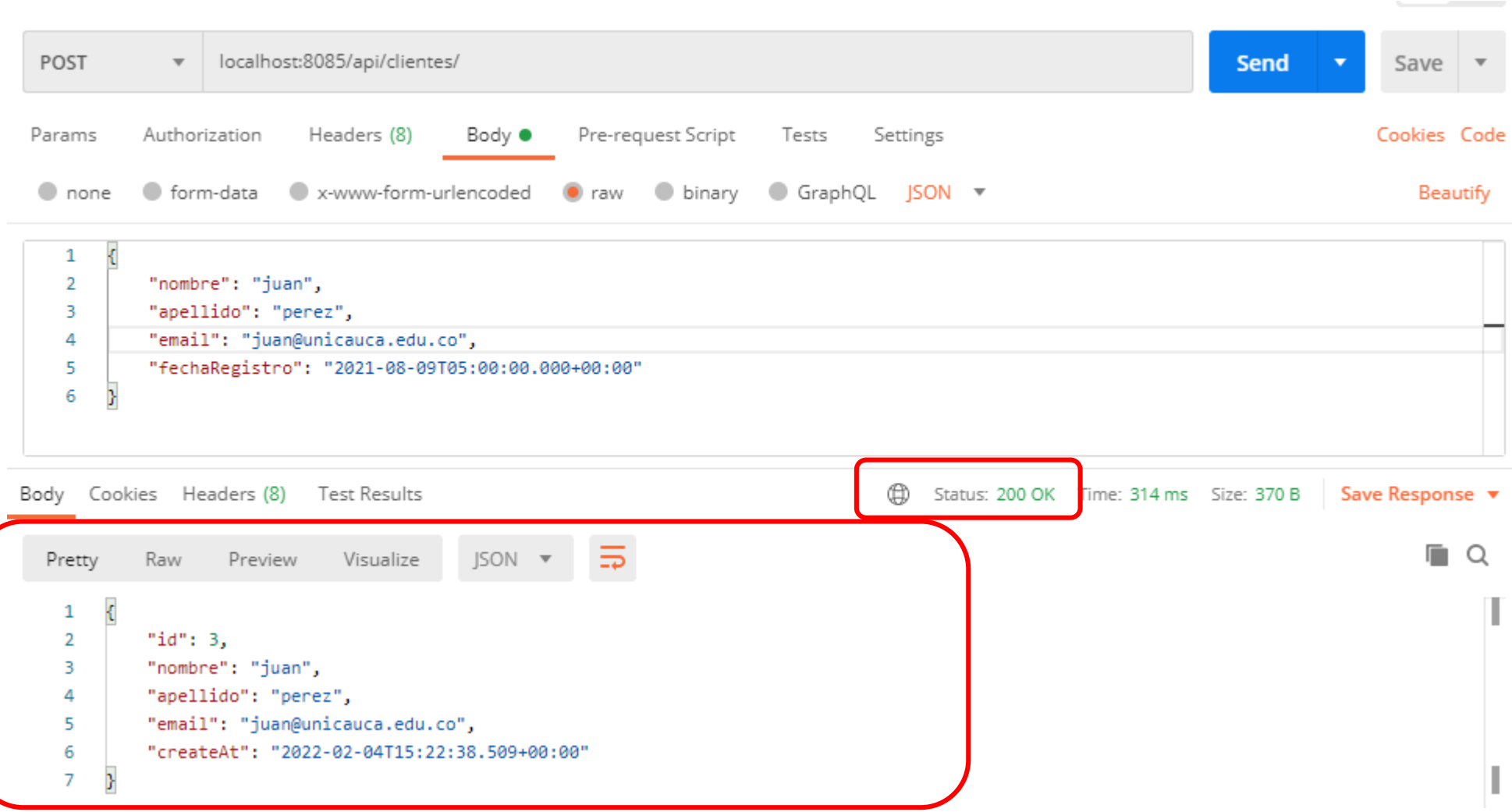


BindingResult

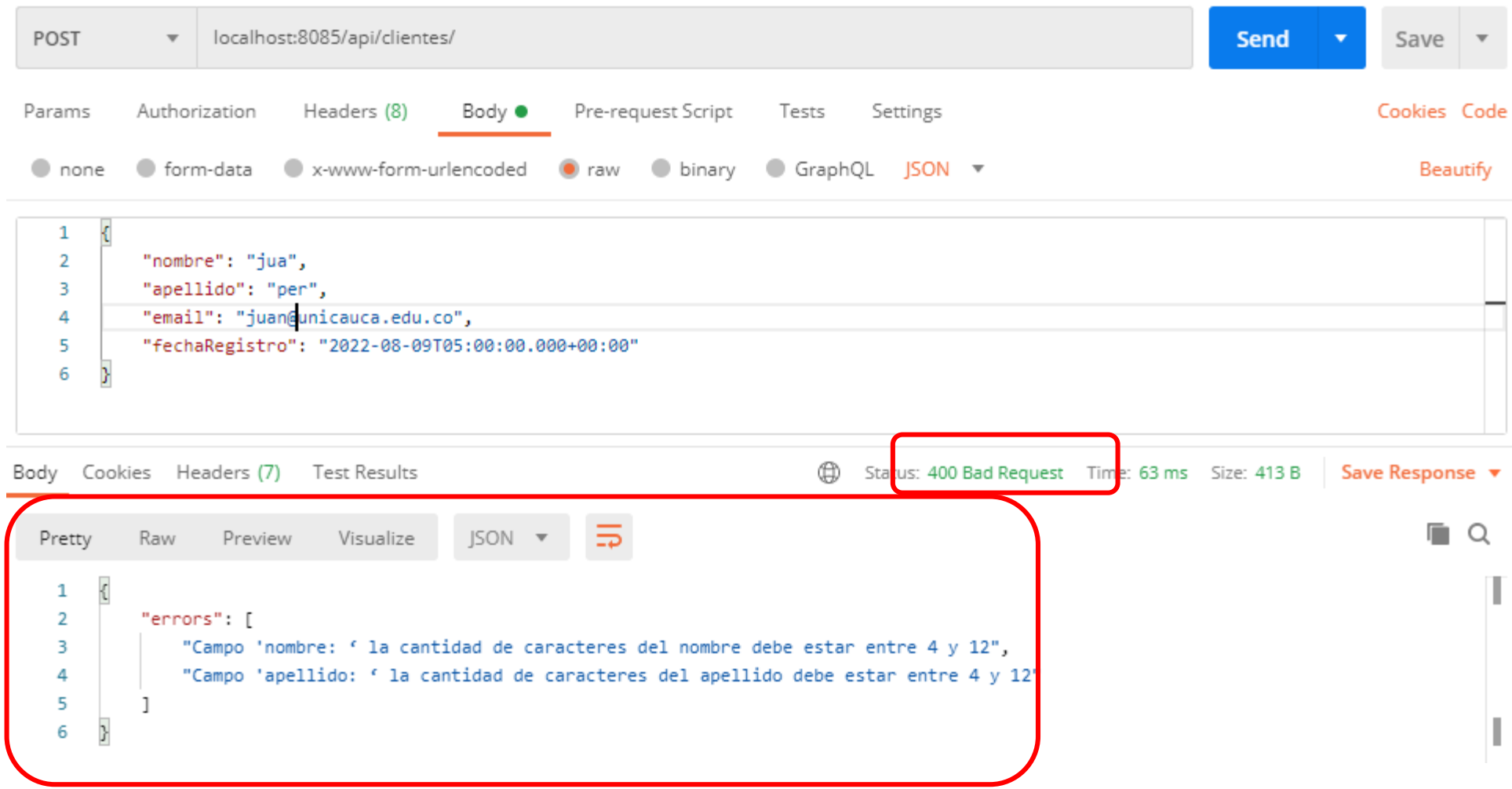
Con **@Valid** se detecta si hay errores en la validación de los datos. Si hay errores el controlador va a retornar los errores y el código 400

```
@PostMapping("/clientes")
public ResponseEntity<?>  create(@Valid @RequestBody Cliente cliente, BindingResult result) {
    cliente.setCreateAt(new Date());
    Map<String, Object> response = new HashMap<>();
    Cliente objCliente;

    if(result.hasErrors())
    {
        List<String> listaErrores= new ArrayList<>();

        for (FieldError error : result.getFieldErrors()) {
            listaErrores.add("Campo '" + error.getField() +": ' "+ error.getDefaultMessage());
        }
        response.put("errors", listaErrores);
        return new ResponseEntity<Map<String, Object>>(response, HttpStatus.BAD_REQUEST);
    }

    try {
        objCliente=this.clienteService.save(cliente);
    }
    catch(DataAccessException e) {
        response.put("mensaje", "Error al realizar la inserción en la base de datos");
        response.put("error", e.getMessage() + "" + e.getMostSpecificCause().getMessage());
        return new ResponseEntity<Map<String, Object>>(response, HttpStatus.INTERNAL_SERVER_ERROR);
    }

    return new ResponseEntity<Cliente>(objCliente, HttpStatus.OK);
}
```

BindingResult



BindingResult



Referencias

https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/constraints/package-summary.html

https://reflectoring.io/bean-validation-with-spring-boot/

https://danielme.com/2018/10/10/validaciones-con-hibernate-validator/

Muchas gracias

Preguntas