15ª Emisión

DIPLOMADO

Desarrollo de Sistemas con Tecnología Java

Módulo 9

Desarrollo de aplicaciones Web con Spring Boot MVC

Mtro. ISC Miguel Ángel Sánchez Hernández

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Dirección General de Cómputo y de Tecnologías de información y Comunicación
Dirección de Docencia en TIC

DGTIC

Educación Continua
1971 - 2021

Objetivo

Ocupar Data Binding, Validaciones y Conversiones en formularios con Spring MVC

Lo que veremos

- Data Binding
- Validaciones
- Crear nuestras validaciones
- Hacer un convertidor

Data Binding

Un Data Binding en Spring MVC es el método donde se extrae dinámicamente la información de un formulario de entrada y se asigna al **Modelo** que estemos ocupando en la aplicación. Lo que tenemos automáticamente es:

- Validar los datos de entrada
- La conversión de tipos
- La información enviada siempre es texto
- @InitBinder

Data Binding

```
public class Usuario{
        private String nombre;
        private String correo
        //get
        //set
}
```

```
<form th:action="@{entrada-uno}"
        method="post">
        <input type="text" name="nombre"/>
        <input type="text" name="correo"/>


</form>
```

```
@PostMapping("/ entrada-uno")
public String salvar(Usuario usuario){
        return "cliente/alta-cliente";
}
```

Validar Datos

Para hacer esta validación ocuparemos el started validation, para esto hay que agregarlo a pow.xml.

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

Las importaciones son ocupando *jakarta.validation.*,* ahora nosotros podemos ocupar anotaciones para validar las entradas del usuario.

Ejercicio 4: Data Binding

Validar Datos y no perder las entradas

```
@PostMapping("entrada-dos")
   public String salvarConParam(@Valid @ModelAttribute(“cliente”)  Cliente cliente,
                                                BindingResult result, Model model){
                        if(result.hasErrors()){
                        }
            …..
}
```

```
<form th:action="@{/comprador/salvar-comprador}" th:object="${cliente}" method="post">
      .
      .
            <label for="nombre">Nombre</label>
            <input type="text" class="form-control" id="nombre" name="nombre"
                        placeholder="Nombre del usuario"
                        th:field="*{nombre}">
```

Validar Datos con Thymeleaf

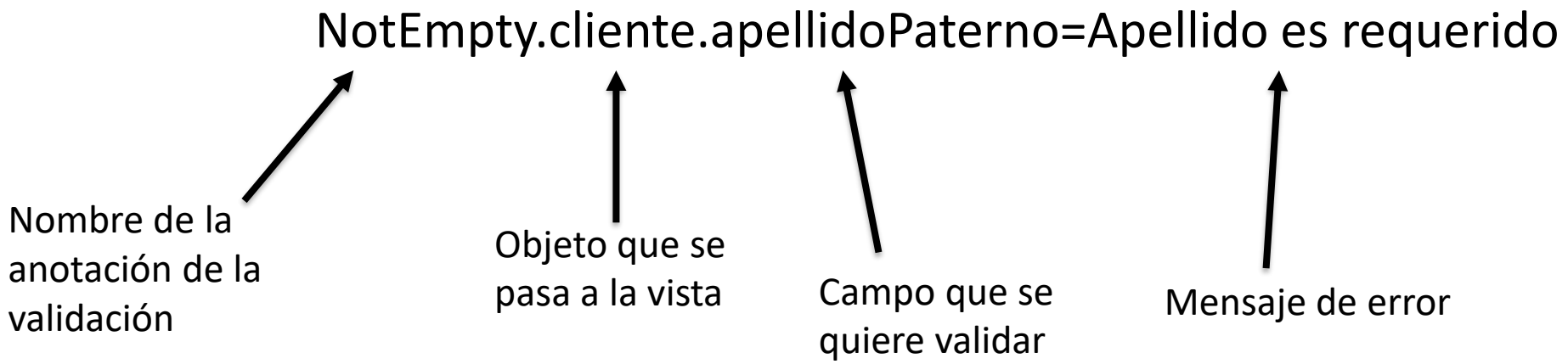
```
<form th:action="@{/cliente/entrada-tres}" th:object="${cliente}" method="post">
        <div th:if="${#fields.hasErrors('*')}" class='alert alert-danger' role="alert">
          Verifique los siguientes errores
          <ul>
            <li th:each="err : ${#fields.errors('*')}" th:text="${err}"/>
          </ul>
        </div>
……
            <div class="form-group">
              <label for="nombre">Nombre</label>
              <input type="text" class="form-control" id="nombre" name="nombre"
                  placeholder="Nombre del usuario"
                  th:field="*{nombre}">
            </div>
```

Personaliza mensaje con anotaciones

```
public class CompradorEntity implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Integer codigo_cpr;
    @NotEmpty(message = "No debe de ser vacío {0}")
    @Column(name = "nombre")
    @Length(min = 1,max = 15)
    private String nombre;
}
```

Personaliza mensaje de errores con properties

Nosotros podemos personalizar los mensajes creando un archivo llamado **messages.properties**, en la carpeta resources.



@Pattern

Nosotros podemos ocupar la anotación @Patten para incrustar una validación a un dato requerido.

```
@Pattern(regexp="55[0-9]{8,8}")
  private String telefono;
```

Pattern.cliente.telefono="Se necesita comenzar con 55 teléfono"

Crear nuestra validación implementando Validator

Para poder implementar nuestra validación en Spring MVC, se necesita importar del paquete de **org.springframework.validation.Validator**, e implementar los dos métodos supports y validate.

```
@Component
public class ClienteValidacion implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
    }

    @Override
    public void validate(Object target, Errors errors) {
    }
}
```

Agregar Validator a @InitBinder

@InitBinder permite crear métodos de configuración al Data Bindig para nuestro controlador, si ocupamos el método setValidator(), descarta las anteriores validaciones que tengamos, para evitar eso ocupamos addValidators().

```
@InitBinder("cliente")
   public void correo(WebDataBinder binder){
      binder.addValidators(clienteValidacion);
   }
```

Validación con anotaciones personalizadas

Para crear nuestras anotaciones personalizadas tenemos que ocupar

- @interface
- Implementar ConstraintValidator

```
@Constraint(validatedBy = NombreClase.class)
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD,ElementType.METHOD})
public @interface NombreClase {
        String message() default “mensaje";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {};
}
```

Validación con anotaciones personalizadas

```
public class NombreClase implements ConstraintValidator<NombreClase,String> {

    @Override
    public boolean isValid(String s, ConstraintValidatorContext constraintValidatorContext) {
            return false;
    }
}
```

Convertidor con Spring MVC

Si necesitamos convertir los datos a un formato en específico podemos extendernos de PropertyEditorSupport. Al último se agrega el convertidor en @InitBinbder.

```
public class MayusculasConverter extends PropertyEditorSupport {
   @Override
   public void setAsText(String text) throws IllegalArgumentException {
      setValue(text.toUpperCase());
   }
}

@InitBinder("comprador")
   public void apellido(WebDataBinder binder){
            binder.registerCustomEditor(String.class,"direccion",
           new MayusculasConverter());
   }
```

Lo que aprendimos

- Ocupar Data Binding
- Validaciones por defecto
- Creación de nuestras propias validaciones
- Creación de un convertidor