



Eusko Jaurlaritzaren Informatika Elkarte  
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

## Gestión de validaciones

Fecha: 18/09/2012

Referencia:

EJIE S.A.  
Mediterráneo, 14  
Tel. 945 01 73 00\*  
Fax. 945 01 73 01  
01010 Vitoria-Gasteiz  
Posta-kutxatila / Apartado: 809  
01080 Vitoria-Gasteiz  
[www.ejje.es](http://www.ejje.es)



UDA – Utilidades de desarrollo de aplicaciones by [EJIE](#) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported License](#).

## Control de documentación

Título de documento: Gestión de validaciones

### Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	22/06/2012	Primera versión.
	1.1.0	18/09/2012	Añadida

### Cambios producidos desde la última versión

### Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma:

Fecha:

Distribución:

### Referencias de archivo

Autor:

Nombre archivo:

Localización:

# Contenido

	Capítulo/sección	Página
1	Introducción	1
2	Arquitectura	2
2.1	<i>Dependencias</i>	3
2.2	<i>Clases Java</i>	3
3	Configuración	4
3.1	Requisitos	4
4	Bean Validation	5
4.1	Hibernate Validator JSR-303	5
4.2	Definición de validaciones	5
4.3	Grupos de validaciones	6
5	Proceso de validación	7
5.1	Validación SpringMVC	7
5.1.1.	Gestión automática de errores de validación	7
5.1.2.	Gestión de errores de validación realizada por el usuario	8
5.1.2.1.	Hibernate Validator JSR-303	8
5.1.2.2.	Validador propio	9
5.2	Comunicación con la capa de presentación	10
5.2.1.	Objeto de transferencia	10
5.2.2.	MessageWriter	11
5.2.3.	ValidationManager	11
6	Migración desde versiones previas a UDA v2.0.0	14
6.1	<i>Clases obsoletas (deprecated)</i>	14
6.2	<i>Nuevas clases</i>	14

	6.3	Proceso de migración	14

## 1 Introducción

En el presente documento se va a detallar el subsistema de validaciones implementado en UDA.

Dicho componente está basado en los conceptos sobre los que se erige el estándar de validaciones de Java Enterprise Edition 6 (JEE6), ya que el diseño de dicho subsistema está alineado con los fundamentos de la Java Specification Request 303 (JSR 303: Bean Validation).

En los siguientes apartados, se detallará la arquitectura del mismo, así como el proceso de configuración y uso.

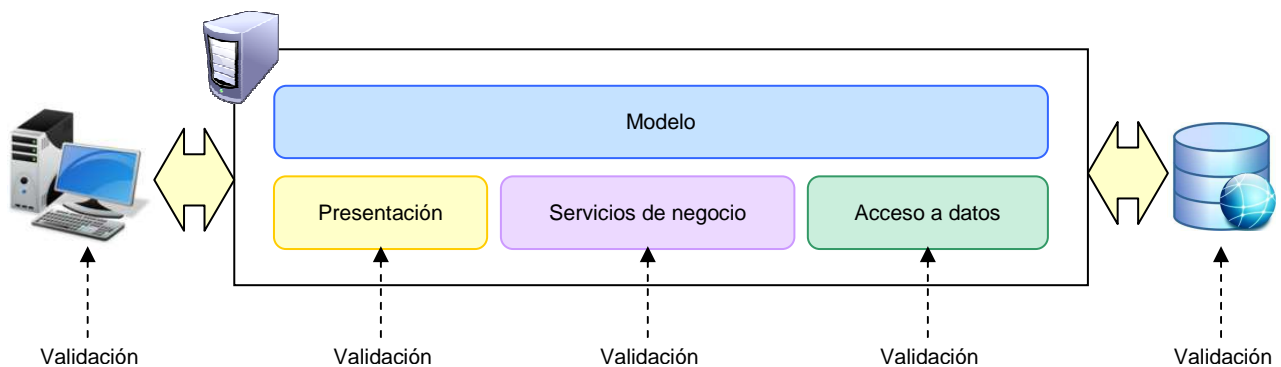
## 2 Arquitectura

Generalmente, los sistemas de validación de datos de las aplicaciones basadas en Java Enterprise Edition (JEE) suelen ser complejos.

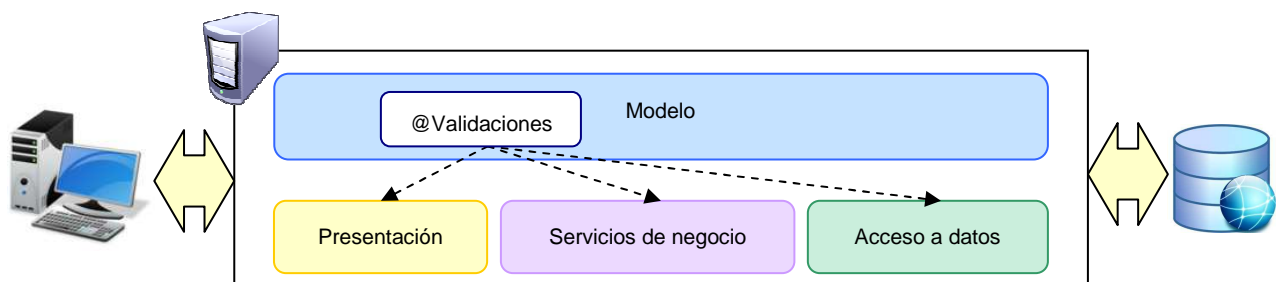
Los principales problemas que se presentan en estos sistemas son:

- Existencia de reglas de validación distribuidas por las diferentes capas de las aplicaciones.
- Las reglas de validación pueden tener diferentes modos de implementación (formato, contenido...) dependiendo de la capa de la aplicación.
- En algunos casos, estas reglas pueden estar definidas en diferentes lenguajes de programación (JavaScript, Java, PL/SQL,...).

Una representación de este enfoque a la hora de crear las reglas de validación sería el siguiente:



El sistema de validación de UDA se basa en el principio de centralizar las reglas de validación en el modelo de datos. De esta manera, se consigue que las reglas sean consistentes y uniformes, ya que se definen una sola vez en toda la aplicación, van directamente ligadas al objetivo de la validación (los datos que contiene el modelo) y son accesibles desde cualquier capa de las aplicaciones JEE.



Para ello, UDA se basa en el estándar JSR-303 Bean Validation el cual define una API y un modelo de metainformación para la validación de entidades. El formato de metainformación usado por defecto son las anotaciones. Estas son añadidas directamente sobre el modelo sus propiedades y/o métodos.

En los siguientes apartados se detallará el modo de utilizar y configurar las anotaciones.



## 2.1 Dependencias

Para poder hacer uso del componente de validaciones es necesario incluir en la aplicación una serie de librerías de las que depende. Para ello, se deberán incluir las siguientes entradas en el *pom.xml* de la aplicación. El fichero se encuentra en <xxx>EAR/pom.xml.

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  ...
  ...
  <properties>
    ...
    <com.ejje.x38.version>2.0.0</com.ejje.x38.version>
  </properties>
  <dependencies>
    ...
    ...
    <!-- JSR 303 with Hibernate Validator -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.2.0.Final</version>
    </dependency>
    ...
    ...
    <dependency>
      <groupId>com.ejje.x38</groupId>
      <artifactId>x38ShLibClasses</artifactId>
      <version>${com.ejje.x38.version}</version>
    </dependency>
    ...
    ...
  </dependencies>
</project>
```

## 2.2 Clases Java

A continuación se detallan las nuevas clases incluidas en x38 para gestionar las validaciones:

-  com.ejje.x38.validation.ValidationController
  - Clase de UDA que se encarga de realizar validaciones de una propiedad de un bean.
-  com.ejje.x38.validation.ValidationManager
  - Clase que se encarga de proporcionar el resto de componentes y métodos que simplifican y facilitan la gestión de validaciones.

## 3 Configuración

La gestión de validaciones se define a nivel de WAR. Por tanto cada aplicación deberá realizar tantas configuraciones como módulos web tenga.

El código necesario para configurar las clases de x38 se define en el fichero *validation-config.xml*:

```
<!-- Proporciona la API de validaciones de UDA -->
<bean id="validationManager" class="com.ejje.x38.validation.ValidationManager" />

<!-- Controller que permite la validación de una propiedad de un bean -->
<bean id="validateController" class="com.ejje.x38.validation.ValidationController" />
```

Mediante estas dos entradas se configuran las clases que son necesarias para la gestión de validaciones.

### 3.1 Requisitos

Para el correcto funcionamiento de la gestión de validaciones es necesario configurar el sistema de excepciones especificado en la versión v2.0.0 de UDA.

Se debe de configurar el *MvcExceptionHandlerConfig* que es el encargado de gestionar las excepciones de validación generadas.



## 4 Bean Validation

En este apartado se va a dar una primera aproximación sobre cómo realizar las validaciones de datos de acuerdo a la especificación JSR-303 Bean Validation. Debido a esto, este apartado no pretende ser un manual de uso de esta especificación, sino que intenta exponer las principales características sobre las que se ha trabajado en el componente de validación de UDA.

Para una comprensión más extensa de la especificación JSR-303 se recomienda la lectura de la documentación existente en la web de Hibernate Validator <http://www.hibernate.org/subprojects/validator/docs>

### 4.1 Hibernate Validator JSR-303

Las operaciones de validación de datos es una tarea muy habitual en las aplicaciones. Estas validaciones pueden llegar a darse en todas las capas de una aplicación, desde la capa web hasta la capa de acceso a datos. Esta situación presenta el problema de que habitualmente se encuentra una implementación diferente en cada una de las capas para llevar a cabo una misma validación.

La especificación JSR-303 define un modelo de metainformación y una API para llevar a cabo la validación de entidades. Este enfoque será aplicado en todas las capas de la aplicación.

La metainformación asociada a las validaciones se incluirá en los modelos como anotaciones asociadas a sus propiedades, métodos y la entidad misma. Debido a que el modelo es la entidad de transferencia común entre las diferentes capas de la aplicación, dispondremos de las reglas de validación definidas en un único lugar de la aplicación.

### 4.2 Definición de validaciones

Las reglas de validación son indicadas mediante anotaciones de Java. Estas anotaciones pueden ser añadidas tanto a la propiedad como a su método getter.

Ejemplo de anotación de validación realizada sobre la propiedad:

```
public class Foo {  
    @NotNull  
    private String bar;  
}
```

Ejemplo de anotación de validación realizada sobre el método getter correspondiente:

```
public class Foo {  
    private String bar;  
  
    @NotNull  
    public getBar(){  
        return this.bar;  
    }  
}
```

También existe la posibilidad de realizar anotaciones de validación a nivel de la clase. Este tipo de validaciones resulta útil cuando la validación depende de la correlación del estado de diferentes variables.

En el siguiente ejemplo se anota la clase para realizar una validación personalizada sobre la clase.

```
@BarCount
public class Foo {
    private List<Bar> bar;
}
```

Mediante anotaciones se puede indicar la validación de una entidad anidada a la entidad principal. Este sería un ejemplo de dos entidades anidadas cuyas propiedades están anotadas con reglas de validación:

```
public class Foo {
    @NotNull
    private String fool;
    private String foo2;
    @NotNull
    @Valid
    private Bar bar;
}

public class Bar {
    @NotNull
    private String bar1;
}
```

De acuerdo a este ejemplo, la validación de un objeto Foo implicará la validación tanto de sus propiedades anotadas como de la validación del objeto Bar que tiene como propiedad.

### 4.3 Grupos de validaciones

Es posible definir grupos de validaciones para acotar las reglas de validación que van a ser tenidas en cuenta durante la validación. Esto permite que se puedan definir diferentes conjuntos de reglas de validaciones para cada diferente necesidad funcional.



En el siguiente ejemplo se definen dos grupos de validaciones:

```
public class Foo {
    @NotNull(groups = ValidationGroupEdit.class)
    private String idFoo;
    @NotNull(groups = {ValidationGroupAdd.class, ValidationGroupEdit.class} )
    private String descFoo;
}
```

Estas interfaces son utilizadas como identificadores de un grupo de validaciones. En este caso se han utilizado dos interfaces:

```
public interface ValidationGroupAdd {}
public interface ValidationGroupEdit {}
```

De este modo estamos definiendo dos grupos de validaciones:

-  *ValidationGroupAdd* solo se aplicará la validación de que la propiedad descFoo no debe ser nula.
-  *ValidationGroupEdit* se aplicarán las validaciones de que las propiedades idFoo y descFoo no deben ser nulas.

## 5 Proceso de validación

### 5.1 Validación SpringMVC

Spring MVC proporciona la posibilidad de realizar una validación del contenido de los objetos sobre los que se realiza el *databinding* de la información enviada desde la capa web.

El componente de Spring se encarga de asociar a las propiedades del bean el valor correspondiente obtenido a partir de la petición. Una vez inicializado el bean se procede a validar el bean de acuerdo a las anotaciones de validación indicadas.

Los beans que deban ser validados se marcarán mediante la anotación `@Validated` del siguiente modo:

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Alumno edit(@Validated(value={ValidationGroupAdd.class}) @RequestBody Alumno
                                alumno){

}
```

En este caso se está indicando que se realice una validación de los datos con los que se ha realizado el *databinding* sobre el bean alumno. En el ejemplo también se puede apreciar como se indican el grupo (o los grupos) de validación que se van a tener en cuenta.

En caso de producirse errores de validación existen diferentes maneras de gestionarlos: de manera automática o gestionada por el desarrollador.

#### 5.1.1. Gestión automática de errores de validación

Desde UDA se proporciona una gestión por defecto de los errores de validación. En caso de producirse errores en las reglas de validación, se produce una excepción que es capturada por el sistema de excepciones.

Dependiendo del tipo de anotación que acompaña a la `@Validated` el tipo de la excepción será diferente:

- En el caso de utilizar `@RequestBody` para realizar el *databinding* se lanza una excepción de tipo **org.springframework.web.bind.MethodArgumentNotValidException**.

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Alumno edit(@Validated @RequestBody Alumno alumno){

}
```

Como ejemplo concreto, la anotación `@RequestBody` se utiliza en las peticiones *application/json* en las que se realiza el *databinding* mediante Jackson.

- En el caso de utilizar `@ModelAttribute` para realizar el *databinding* se lanza una excepción de tipo **org.springframework.validation.BindException**.

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Alumno edit(@Validated @ModelAttribute Alumno alumno){

}
```

Como ejemplo concreto, la anotación `@ModelAttribute` se utiliza en las peticiones *multipart/form-data* en las que se realiza el envío de ficheros y de parámetros de manera conjunta.

Estas excepciones son capturadas por el sistema de excepciones que las procesa y se genera un objeto JSON de error que se envía a la capa de presentación para mostrarse al usuario. La petición devuelve un código de error HTTP 406.

El objeto JSON enviado como respuesta tiene la siguiente estructura:

```
{
  "rupErrorFields": {
    "id": ["Campo obligatorio"],
    "nombre": ["Campo obligatorio"],
    "apellido1": ["Longitud máxima 50 caracteres"]
  }
}
```

Dentro de la estructura JSON enviada se crea un objeto de nombre *rupErrorFields* en el cual se almacena para cada campo, o identificador del elemento, la lista de errores generados en el proceso de validación.

### 5.1.2. Gestión de errores de validación realizada por el usuario

Hay ocasiones en las que las necesidades funcionales de la aplicación no pueden ser cubiertas mediante el sistema automático de gestión de las validaciones.

Por ejemplo puede ser necesario realizar validaciones extra, por ejemplo comprobar que el nombre de usuario introducido sea único. Para estas situaciones se han identificado una serie de escenarios.

#### 5.1.2.1. Hibernate Validator JSR-303

Las validaciones mediante Hibernate Validator se pueden realizar de manera manual sin utilizar la integración con el mismo que ofrece SpringMVC. De este modo podemos realizar el mismo tipo de validaciones basado en anotaciones en el model pero teniendo un mayor control sobre la gestión de los errores.

A continuación se muestra un ejemplo de validación realizado en un controller mediante el uso directo del validador definido en la configuración de la aplicación. Los errores de validación se presentan mediante un *Set<ConstraintViolation<T>>*.

```
@Controller
@RequestMapping (value = "/administracion/alumno")
public class AlumnoController {

    @Autowired
    private Validator validator;

    @RequestMapping(method = RequestMethod.PUT)
    public @ResponseBody Alumno edit(@RequestBody Alumno alumno){
        Set<ConstraintViolation<Alumno>> violations = validator.validate(alumno,
            ValidationGroupAdd.class);

        // Gestión de los errores de validación
    }
}
```

Para facilitar la gestión posterior de los errores de validación (la gestión del set de *ConstraintViolation* no es trivial) se ha incluido un método en la clase *ValidationManager* proporcionada por la librería x38. Este método realiza la validación haciendo uso del mismo validador, pero proporciona los errores mediante un objeto de tipo *org.springframework.validation.Errors*. A continuación se detalla un ejemplo.

```
@Controller
@RequestMapping (value = "/administracion/alumno")
public class AlumnoController {

    @Autowired
    private ValidationManager validationManager;

    @RequestMapping(method = RequestMethod.PUT)
    public @ResponseBody Alumno edit(@RequestBody Alumno alumno){

        Errors errors = new BeanPropertyBindingResult(alumno, "alumno");
        validationManager.validate(errors, alumno, ValidationGroupAdd.class);

        // Gestión de los errores de validación
    }
}
```

Cabe destacar que existe un modo de realizar la validación del bean mediante SpringMVC y el uso de la anotación `@Validated` y gestionar manualmente los errores de validación. Añadiendo un parámetro de tipo `Errors` o `BindingResult` después del parámetro sobre el que se realiza el *databinding* se añadirán en dicho objeto los errores de validación producidos en vez de lanzar una excepción.

**NOTA:** Debido a ciertas limitaciones que presenta Spring en este aspecto, únicamente es posible utilizar esta solución cuando se use la anotación `@ModelAttribute` para realizar el *databinding*. En caso de utilizar la anotación `@RequestBody` se producirá una excepción en el despliegue de la aplicación.

Un ejemplo de uso de esta funcionalidad es el siguiente:

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Alumno edit(@Validated @ModelAttribute Alumno alumno, Errors errors){

}
```

### 5.1.2.2. Validador propio

En caso de que no se desee realizar las validaciones mediante anotaciones o estas no lo permitan, o se necesite de validaciones adicionales, es posible implementar validadores propios.

Para implementar un validador propio se deberá crear una clase que extienda de `org.springframework.validation.Validator` y que implemente los métodos *supports* y *validate*.

A continuación se muestra el código de un ejemplo de validador:

```
public class AlumnoValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return clazz.equals(Alumno.class);
    }

    @Override
    public void validate(Object obj, Errors errors) {

        Alumno alumno = (Alumno)obj;
        if (alumno!=null){
            if (alumno.getPais()!=null && "108".equals(alumno.getPais().getId())){
                if (alumno.getAutonomia()!=null || alumno.getAutonomia().getId()== null
                || "".equals(alumno.getAutonomia().getId())){
                    errors.rejectValue("autonomia.id", "validacion.required");
                }
            }
        }
    }
}
```

```

    }else{
        if (alumno.getDireccion()==null || "".equals(alumno.getDireccion())){
            errors.rejectValue("autonomia.id", "direccion");
        }
    }
}
}
}
}

```

En el ejemplo se muestra una validación en la cual dependiendo del valor que contiene el bean alumno se comprueban diferentes valores del bean. El validador se utilizaría del siguiente modo desde una clase java:

```

AlumnoValidator alumnoValidator = new AlumnoValidator();
alumnoValidator.validate(alumno, errors);

```

## 5.2 Comunicación con la capa de presentación

Una vez realizado el proceso de validación y han sido obtenidos los errores de las reglas de validación, el siguiente paso es presentar estos errores al usuario. Para este cometido se ha tratado de simplificar:

- Proceso de generar los errores a enviar a la capa web.
- Estructura del objeto que contiene la información a enviar.
- Proceso de presentar los errores al usuario a partir del objeto recibido por la capa web.

### 5.2.1. Objeto de transferencia

El envío de los errores a la capa de presentación se realiza mediante un objeto JSON con la siguiente estructura:

```

{
  "rupErrorFields":{
    "id":["Campo obligatorio"],
    "nombre":["Campo obligatorio"],
    "apellidos":["Longitud máxima 50 caracteres"]
  },
  "rupFeedback":{
    "Este es un ejemplo de errores enviados desde el servidor",
    "Campo obligatorio",{
      "label":"Prueba de estilos",
      "style":"rojo"},
    [
      "Nivel 1.1",
      "Nivel 1.2",
      "Nivel 1.3"
    ]
  ]
}

```

El objeto *rupErrorFields* contiene, para cada campo o identificador del elemento, la lista de errores generados en el proceso de validación.

El objeto *rupFeedback* contiene una estructura JSON que puede ser interpretada por el componente RUP *feedback*. En la guía de uso de este componente se amplía la información referente a este aspecto.

### 5.2.2. MessageWriter

La clase *MessageWriter* es una clase de ayuda para generar mensajes a visualizar en los componentes *feedback* de la capa web. Esta clase se proporciona mediante la librería x38.

A continuación se muestra un ejemplo de uso de la clase en el que se utilizan todas las funcionalidades que permite:

```
MessageWriter messageWriter = new MessageWriter();

// Inicio de la lista de mensajes
messageWriter.startMessageList();
// Escritura de un mensaje simple
messageWriter.addMessage("Mensaje simple");
// Escritura de un mensaje internacionalizado
messageWriter.addMessage(messageSource, "validacion.required");
// Escritura de un mensaje con estilos aplicados
messageWriter.addComplexMessage("Prueba de estilos", "rojo");
// Escritura de un mensaje internacionalizado con estilos aplicados
messageWriter.addComplexMessage(messageSource, "validacion.required", "rojo");
// Inicio de un subnivel para añadir mensajes anidados
messageWriter.startSubLevel();
// Escritura de varios mensajes
messageWriter.addMessage("Nivel 1.1", "Nivel 1.2", "Nivel 1.3");
// Fin del subnivel
messageWriter.endSubLevel();
// Fin de la lista de mensajes
messageWriter.endMessageList();

// Se obtiene un objeto JSON que contiene una representación de la lista de mensajes
JSONObject jsonObj = messageWriter.getJsonObject();
```

### 5.2.3. ValidationManager

Para facilitar la creación del objeto JSON que se va a enviar a la capa web, se proporcionan una serie de métodos a través de la clase *ValidationManager* incluida en la librería x38.

Los métodos implementados son:

- `Map<String,List<String>> errorsMap = validationManager.getErrorsAsMap(Errors errors);`

Genera a partir de unos errores indicados como parámetros, un mapa en el cual cada elemento está formado del siguiente modo:

- El *key* del elemento del mapa está formado por el nombre de la propiedad sobre la que se ha producido el error de la validación o por el identificador que se ha asignado a la validación.
- El objeto asociado al *key* está formado por una lista de mensajes de error internacionalizados a partir del *key* de error.

- `List<String> errorsList = validationManager.getErrorsAsList(Errors errors);`

Genera a partir de unos errores indicados como parámetros, una lista formada por los mensajes de error internacionalizados.

- `Map<String,Object> mapa = validationManager.getRupFeedbackMsg (Errors errors);`

Genera a partir de un mensaje y un estilo, una estructura para devolver una estructura que pueda ser convertida a formato JSON e interpretada por el feedback.

- *msg*: Mensaje que debe ser visualizado por el feedback.

- style: Estilo que se debe mostrar con el feedback.

```
• JSONObject jsonObject = validationManager.getMessageJSON(Object fieldErrors, Object feedbackMessage, String style);
```

Genera a partir de una serie de parámetros, un objeto JSON con el formato específico para ser interpretado por los componentes RUP de la capa web.

- fieldErrors: Errores de validación que serán incluidos en el objeto JSON *rupErrorFields*.
- feedbackMessage: Mensaje que debe ser visualizado por el feedback.
- style: Estilo que se debe mostrar con el feedback.

Un ejemplo de uso de estos métodos sería el siguiente:

```
@RequestMapping(value = "ejemplo", method = RequestMethod.POST, produces="application/json")
public @ResponseBody Object ejemplo(@RequestBody Alumno alumno, Model model, HttpServletRequest
    request, HttpServletResponse response) {
    Errors errors = new BeanPropertyBindingResult(alumno, "alumno");
    validationManager.validate(errors, alumno, AlumnoEjemplo2Validation.class);

    /*
     * Se realiza la siguiente validacion sobre el bean:
     * Se comprueba que el nombre de usuario introducido no
     */
    UserNameValidator usuarioValidator = new UserNameValidator();
    usuarioValidator.validate(alumno, errors);

    /*
     * EJEMPLO DE ENVIO DE MENSAJES DESDE EL SERVIDOR
     */
    if (errors.hasErrors()){
        try {
            Map<String, List<String>> fieldErrors = validationManager.getErrorsAsMap(errors);
            response.sendError(406, validationManager.getMessageJSON(fieldErrors).toString());
        } catch (IOException e) {
            // Gestión del error
        }
        return null;
    }

    MessageWriter messageWriter = new MessageWriter();

    // Inicio de la lista de mensajes
    messageWriter.startMessageList();
    // Escritura de un mensaje simple
    messageWriter.addMessage("Mensaje simple");
    // Escritura de un mensaje internacionalizado
    messageWriter.addMessage(messageSource, "validacion.required");
    // Escritura de un mensaje con estilos aplicados
    messageWriter.addComplexMessage("Prueba de estilos", "rojo");
    // Escritura de un mensaje internacionalizado con estilos aplicados
    messageWriter.addComplexMessage(messageSource, "validacion.required", "rojo");
    // Inicio de un subnivel para añadir mensajes anidados
    messageWriter.startSubLevel();
    // Escritura de varios mensajes
    messageWriter.addMessage("Nivel 1.1", "Nivel 1.2", "Nivel 1.3");
    // Fin del subnivel
    messageWriter.endSubLevel();
    // Fin de la lista de mensajes
    messageWriter.endMessageList();

    return messageWriter.toString();
}
```



**IMPORTANTE:** En el caso de utilizar el navegador Internet Explorer 8, la subida de ficheros mediante un formulario se realiza mediante el uso de iframe. Esto es debido a que la subida de ficheros mediante peticiones AJAX no está soportada en este navegador. En estos casos la gestión de errores de validaciones realizadas en el servidor de aplicaciones, no puede hacer un uso correcto de los códigos de error http para la gestión de los mismos.

La configuración que se ha de realizar para permitir la interacción correcta entre los iframes y el resto de la infraestructura (request mappings, http error code, validaciones...) se detalla en el anexo Anexo-Emulacion xhr iframes.doc

## 6 Migración desde versiones previas a UDA v2.0.0

En el presente apartado se va a explicar los pasos a seguir para realizar las modificaciones necesarias en el código de una aplicación en la que se desea migrar del sistema de validaciones implementados en versiones anteriores, al incorporado en la versión v2.0.0.

Las aplicaciones desarrolladas con UDA contaban con una serie de componentes por defecto proporcionados mediante la librería x38. Estos componentes consistían en un filtro y un *servlet* que eran los encargados de gestionar las validaciones realizadas en el servidor a partir de los datos enviados mediante peticiones AJAX.




El sistema de validaciones incorporado en la nueva versión de UDA es retrocompatible con las implementaciones existentes que hacen uso de versiones previas.

En el caso de querer beneficiarse de las mejoras y nuevas funcionalidades de que dispone la nueva versión, se deberán realizar una serie de modificaciones en las aplicaciones.

Para poder llevar a cabo la migración al nuevo sistema de validaciones es necesario realizar previamente la migración al nuevo sistema de excepciones implementado en la versión v2.0.0 de UDA. El proceso de migración se detalla en el documento **Anexo - Gestión de excepciones.doc**.


### 6.1 Clases obsoletas (deprecated)

La siguiente lista contiene las clases de la librería x38 que se han quedado obsoletas con la nueva gestión y por tanto se han definido como *deprecated*:

-  com.ejje.x38.validation.ValidationFilter
-  com.ejje.x38.validation.ValidationRequestWrapper
-  com.ejje.x38.validation.ValidationServlet

### 6.2 Nuevas clases

A continuación se detallan las nuevas clases incluidas en x38 para gestionar las validaciones:

-  com.ejje.x38.validation.ValidationController
  - Clase de UDA que se encarga de realizar validaciones de una propiedad de un bean.

### 6.3 Proceso de migración

A continuación se van a detallar los pasos a seguir para realizar la migración del código y que haga uso de la nueva configuración:

1. En la versión v2.0.0 de UDA se elimina el uso de las clases ValidationFilter y ValidationServlet. Se deberá modificar el fichero <xxxYYY>War/WebContent/WEB-INF/spring/validation-config.xml para **eliminar** la siguiente configuración:

```
...  
...  
  
<!-- Filter that Validates Required Request Bodies -->  
<bean id="validationFilter" class="com.ejje.x38.validation.ValidationFilter">  
    <property name="validationManager" ref="validationManager"/>  
    <property name="localeResolver" ref="localeResolver" />  
</bean>
```

```
<!-- Servlet that Validates Required Request Parameters -->
<bean id="validationServlet" class="com.ejje.x38.validation.ValidationServlet">
    <property name="validationManager" ref="validationManager"/>
    <property name="localeResolver" ref="localeResolver" />
</bean>
...
...
```

2. Añadir en el fichero <xxxYYY>War/WebContent/WEB-INF/spring/validation-config.xml la siguiente entrada:

```
...
...
<bean id="validateController" class="com.ejje.x38.validation.ValidationController" />
...
...
```

3. Eliminar del fichero <xxxYYY>War/WebContent/WEB-INF/web.xml las siguientes líneas:

```
...
...
<!-- Validates data transparently -->
    <filter>
        <filter-name>validationFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>validationFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

...
...
<!-- Handles data validation requests -->
    <servlet>
        <servlet-name>validationServlet</servlet-name>
        <servlet-
class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>validationServlet</servlet-name>
        <url-pattern>/validate/*</url-pattern>
    </servlet-mapping>...

...
...
```

4. Añadir en el fichero <xxxYYY>War/WebContent/WEB-INF/spring/mvc-config.xml la declaración del validador que va a utilizar Spring MVC.

```
...
...
<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" >
    <property name="validationMessageSource" ref="messageSource"/>
</bean>
...
...
```

5. En el mismo fichero, añadir la referencia del validador al *requestMappingHandlerAdapter*. El código a añadir se muestra resaltado.

```
...
...
<bean id="requestMappingHandlerAdapter"
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
            <property name="conversionService" ref="conversionService" />
        </bean>
    </property>
</bean>
```

```
        <property name="validator" ref="validator" />
    </bean>
</property>
<property name="messageConverters">
    <list>
        <bean class="com.ejje.x38.serialization.UdaMappingJacksonHttpMessageConverter">
            <property name="jacksonJsonMapper" ref="jacksonJsonMapper" />
        </bean>
    </list>
</property>
</bean>
...
...
```

6. Por último, se deberán modificar en los controller los métodos en los que se debe de realizar la validación de los datos. Se deberá de añadir la anotación `@Validated` al parámetro del método sobre el que se deben de validar los datos que contiene después de realizarse el *databinding* de los mismos.

```
...
...
...
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Alumno edit(@Validated @RequestBody Alumno alumno){
}
...
...
```