Lista de anotaciones JPA

En este documento se explicará con detalle cuando y como se utilizan las anotaciones JPA necesarias en cualquier proyecto de la asignatura.

**Pregunta**: ¿Dónde podemos utilizar las anotaciones JPA?

**Respuesta**: En el dominio, repositorios, servicios, test, controladores… Dependiendo de dónde y en qué caso nos encontremos usaremos una serie de anotaciones u otras.

# Restricciones del modelo de dominio

Las **restricciones** son las limitaciones de nuestro modelo conceptual y se pueden aplicar a todos los atributos que hayamos definido. Se utilizarán en los métodos getter del dominio. Algunas de ellas son:

|  |
| --- |
| Restricciones para cualquier tipo de dato |
| @NotNull: índica que el atributo no puede ser nulo.  Consejo: utilizaremos @NotNull en una colección cuando:   * Si la colección puede estar vacía (multiplicidad [0…\*]): utilizamos la anotación @NotNull. * Si la colección no puede estar vacía (multiplicidad [1…\*], [5…\*], …): utilizamos la anotación @NotEmpty.   Motivo: debemos reservar un espacio de memoria para una colección, aunque la lista esté inicialmente vacía. |
| @Valid: valida datos de forma recursiva. Normalmente se aplica a tipos de datos definidos por el usuario. De forma más coloquial, se dirige a la clase destino y comprueba que todos sus atributos cumplen con sus restricciones. También se utiliza en los controladores.  Ejemplo: tenemos una entidad *Rendezvous* con un atributo de tipo *GPSPoint*. ¿Cómo podemos validar desde la entidad *Rendezvous* las restricciones de *GPSPoint*? Utilizando la anotación @Valid.    Al validar el atributo GPSPoint de una instancia de *Rendezvous* y encontrar la etiqueta @Valid, validamos a su vez todas las restricciones definidas en. *GPSPoint*    Por regla general, utilizaremos esta etiqueta con aquellos tipos definidos por nosotros.  Consejo: debemos utilizarlas siempre en las relaciones porque nunca vamos a encontrar que una relación sea de tipo Collection<String>. |
| @Size(min = xxx, max=yyy): indica la longitud que puede tener una colección. Pueden indicarse ambos valores o uno de los dos. |
| @ElementCollection: se utiliza para colecciones de datatypes |

|  |
| --- |
| Restricciones para fechas – Tipo Date |
| @Past: la fecha debe ser pasado |
| @Future: la fecha debe ser futura. Esta anotación NO PODEMOS UTILIZARLA.  Supongamos que tenemos una instancia de una entidad cualquiera que tiene un atributo de tipo fecha con esta anotación. Llegará un momento determinado en el que esa fecha deje de ser una fecha futura y pase a ser una fecha pasado. Cuando intentemos editar dicho objeto por alguna razón, al actualizar el objeto de la base de datos el dominio dará un error porque ya es una fecha pasada independientemente de que hubiese sido futura en algún momento.  Además, esto da problemas al poblar la base de datos. Supongamos que creamos el PopulateDatabase.xml en un momento dado y hemos creado para un atributo con la anotación @Future de una entidad concreta con valores en pasado y en futuro para tener varios casos de ejemplo. ¿Qué ocurrirá? Pues saltará un error porque la clase PopulateDatabase.java utiliza Hibernate para realizar la persistencia y, por lo tanto, todos aquellos datos que se incluyan en la base de datos deben cumplirlas. Entonces, por culpa de esta etiqueta, no podríamos añadir valores pasados de una entidad concreta.  Solución: realizar las comprobaciones correspondientes en los servicios a la hora de hacer la persistencia. Podemos usar los siguientes métodos de java.util.Date (<https://docs.oracle.com/javase/7/docs/api/java/util/Date.html>) entre otros:   * boolean after(Date fecha): prueba si la fecha es posterior a la especificada. * Boolean before(Date fecha): prueba si la fecha es anterior a la especificada. |
| @Temporal(…): indica cómo persiste una fecha en la base de datos.  Una instancia de tipo Date almacena una fecha y una hora, pero podemos encontrar casos en los que no es necesario almacenar la hora, o la fecha. Dependiendo de lo que queramos persistir en la base de datos:   * @Temporal(TemporalType.DATE): solo almacena la fecha. * @Temporal(TemporalType.TIME): solo almacena la hora. * @Temporal(TemporalType.TIMESTAMP): almacena la fecha y el tiempo. |
| @DateTimeFormat(pattern = “[patron]”): formatea una fecha  Error común: Un problema habitual con las fechas es que hay demasiados formatos para representarlas. Por ejemplo, una fecha como “1 de febrero de 2003” puede representarse como “01/02/03”, “01/02/2003”, etc. Esta etiqueta toma un parámetro patrón que indica dicha interpretación. Dicho patrón puede ser algunos de los planteados en <https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html> o personalizado. En la lección L03, las fechas las añadimos con el formato “yyyy/MM/dd HH:mm”. El problema es que hay una interferencia y debemos utilizar “dd/MM/yyyy HH:mm” o de lo contrario, las fechas serán malinterpretadas.  @DateTimeFormat(pattern = “dd/MM/yyyy HH:mm”)  Indica cómo se va visualizar la fecha en los comentarios.  Aclaración: a medida que hemos trabajado con las fechas que usan estas anotaciones, nos hemos dado cuenta de que la anotación @DateTimeFormat no restringe la fecha a la hora de POR TERMINAR |

|  |
| --- |
| Restricciones para colecciones – Tipo Collection |
| @NotEmpty: la colección no debe estar vacía.  Error común: esta etiqueta implica de forma implícita @NotNull, es decir, si estamos en un atributo de tipo Collection<[Entidad]> y debe tener algún dato, no añado la etiqueta @NotNull. |

|  |
| --- |
| Restricciones para números – Tipo Integer, int, Double double, Float float, … |
| @Min(xxx): debe ser un número menor o igual que xxx. El valor xxx debe ser un número de tipo entero. Por ejemplo, si queremos que el mínimo valor de un atributo no sea menor o igual que 0 en un atributo de tipo double, lo ideal es que utilicemos la anotación @DecimalMin:  @DecimalMin(value = “0”) |
| @Max(xxx): debe ser un número mayor o igual que xxx. El valor xxx debe ser un número de tipo integer. Por ejemplo, si queremos que el máximo valor de un atributo no sea mayor o igual que 50 en un atributo de tipo double, lo ideal es que utilicemos la anotación @DecimalMax:  @DecimalMax(value = “50”) |
| @DecimalMin(value = “xxx”) |
| @Range(min = (tipoBasico) xxx, max = (tipoBasico) yyy): debe ser un número en el intervalo [xxx,yyy] |
| @Digits(integer, fraction): debe ser un número dentro de un rango especificado, donde:   * integer: especifica la cantidad máxima de dígitos de todo el número con el que estemos trabajando. * fraction: especifica los dígitos correspondientes a fracciones (los decimales)   Ejemplo: si el tipo de número con decimales con el que vamos a trabajar es, por ejemplo, 1934’23, la anotación quedaría de la siguiente manera:  @Digits(integer = 6, fraction = 2) |
| @CreditCardNumber: debe ser lexicamente/sintácticamente un número de tarjeta de crédito válida.  Error común: supongamos que tenemos un datatype de tipo CreditCard, esta tendrá una serie de atributos. Por lógica, está anotación debe ir sobre el método getNumber() de la entidad o datatype CreditCard.    ¿Dónde NO debemos usar esta etiqueta? En aquellos sitios donde se utilice el tipo CreditCard porque esta etiqueta ya se encuentra definida en el atributo correspondiente de CreditCard. |
| @Pattern(regex =”[patrón]”): |

|  |
| --- |
| Restricciones para cadenas – Tipo String |
| @Length(min,max): la longitud de la cadena debe ser un número en el intervalo [min, max] |
| @NotBlank: indica que una cadena no debe ser nulo, una cadena vacía o una cadena compuesta por valores en blanco exclusivamente.  Error común: esta etiqueta implica de forma implícita @NotNull, es decir, si estamos en un atributo de tipo String que no puede ser nulo, no añado la etiqueta @NotNull porque va implícito en @NotBlank.  Pregunta: si un atributo es opcional, ¿podemos utilizar la anotación @NotBlank?  Respuesta: no, porque la anotación @NotBlank implica @NotNull. |
| @Email: la cadena debe ser lexicamente/sintácticamente una dirección de correo válida.  No funciona del todo bien, pero si un atributo del dominio es un email, añadimos esta anotación y listo. |
| @Url: la cadena debe ser lexicamente/sintácticamente una dirección url válida.  No funciona del todo bien, pero si un atributo del dominio es una url, añadimos esta anotación y listo. |
| @SafeHtml (whitelistType = Enum): controla que el texto enriquecido proporcionado por el usuario no contenga código HTML inseguro, como bloques ECMA Script, etc. Se utilizan para evitar ataques de tipo Cross Scripting (XSS) que se explicarán en el segundo cuatrimestre.  Consejo: si necesitamos especificar el código HTML que se puede ejecutar, podemos añadir whitelistType como parámetro de entrada a la anotación ([ver concepto de lista blanca](https://es.wikipedia.org/wiki/Lista_blanca)), pudiéndole asignar los siguientes valores utilizando un enumerado:   * WhiteListType.BASIC: esta lista blanca permite una gama más completa de nodos de texto: a, b, blockquote, br, cite, code, dd, dl, dt, em, i, li, ol, p, pre, q, small, strike, strong, sub, sup, u, ul. * WhiteListType.BASIC\_WITH\_IMAGES: esta lista blanca permite las mismas etiquetas de texto que BASIC, y también permite img (con el parámetro src apuntando a http o https). * WhiteListType.NONE: esta lista blanca sólo permite nodos de texto: todo el código HTML se eliminará. * WhiteListType.RELAXED: esta lista blanca permite una gama completa de texto y una estructura HTML: a, b, blockquote, br, caption, cite, code, col, colgroup, dd, dl, dt, em, h1, h2, h3, h4, h5, h6, i, img, li, ol, p, pre, q, small, strike, strong, sub, sup, table, tbody, td, tfoot, th, thead, tr, u, ul. * WhiteListType.SIMPLE\_TEXT: esta lista blanca permite sólo el formato de texto simple: b, em, i, strong, u.   1º Recomendación: para evitar posibles inconvenientes, utilizaremos la opción NONE.  2º Recomendación: para evitar posible problemas de hacking, siempre que nos encontremos con un atributo de tipo String, añadiremos la siguiente anotación.  @SafeHtml (whitelistType = WhiteListType.NONE)  Nota: Si no se especifica nada, se utiliza la opción RELAXED. Para más información, consultar los siguientes enlaces: [enlace1](https://docs.jboss.org/hibernate/validator/4.2/api/org/hibernate/validator/constraints/SafeHtml.html), [enlace2](https://docs.jboss.org/hibernate/validator/4.2/api/org/hibernate/validator/constraints/SafeHtml.WhiteListType.html), [enlace3](https://stackoverflow.com/questions/42360874/java-hibernate-safehtml-not-allows-url-links), [enlace4](https://stackoverflow.com/questions/32052495/spring-defaulthtmlescape-doesnt-prevent-xss-attack). |

# Anotaciones en los convertidores

Los **convertidores (*converters*)** son clases que ayudan a transformar una entidad en una representación como String mediante su identificador que es fácil de incrustar en HTML o viceversa.

Para este tipo de clases debemos incluir las siguientes anotaciones:

|  |
| --- |
| Anotaciones en los convertidores |
| @Component: indica que el convertidor es una clase/componente de Spring, permitiendo considerarlo como uno de sus objetos. Se añade en la cabecera de la clase. |
| @Transactional: indica que los métodos incluidos en el convertidor se van a ejecutar en el contexto de una transacción. Se añade en la cabecera de la clase. |

**Importante**: hay que incluirlo en los convertidores tanto de entidades como de datatypes.

Ejemplo – de objeto a String:

**@Component**

**@Transactional**

public class [Entidad/Datatype]ToStringController implements

Converter<[Entidad/Datatype]/String> {

…

}

Ejemplo – de String a objeto:

**@Component**

**@Transactional**

public class StringTo[Entidad/Datatype]Controller implements

Converter<String/[Entidad/Datatype]> {

…

}

# Anotaciones en repositorios

Un **repositorio (*repository*)** es una interfaz mediante la cual podemos persistir un tipo de entidades de dominio en una base de datos relacional.

Para este tipo de clases debemos incluir las siguientes anotaciones:

|  |
| --- |
| Anotaciones en los repositorios |
| @Repository: es una especialización de la anotación @Component. Indica que el repositorio es una clase/componente de Spring, permitiendo considerarlo como uno de sus objetos. Se añade en la cabecera de la clase. |
| @Query: anotación de Spring Data JPA que permite al usuario especificar una consulta JPQL que debe ejecutarse cada vez que se llama al método al que está asociado. Los parámetros se introducen como variables de posición como: ?1, ?2, ?3, …  Error común: supongamos que hemos creado una query que necesitamos para el proyecto y la incluimos dentro de la anotación. Acto seguido, arrancamos el servidor y no se lanza la aplicación correctamente, dando como respuesta una serie de errores de difícil interpretación. El motivo puede ser que la query que hemos definido sea incorrecta por diversos motivos (sintaxis incorrecta debido cambios en el modelo de dominio Java, la query no se ha probado antes de incluirse en el repositorio, etc…).  Solución: para evitarlo, antes de incluir una query en el repositorio realizaremos las siguientes comprobaciones:   * Probar la query mediante la clase creada por los profesores QueryDatabase.java * La query, a diferencia de cuando se utilizó en QueryDatabase.java, no debe tener el carácter “;” al final. * Verificar que los parámetros de entrada de la query están bien incluidos y corresponden con cada uno de los parámetros de entrada del método asociado.   Error común: cuando necesitamos pasar un parámetro de tipo enumerado, debemos tener cuidado en cómo lo pasamos debido a que puede ocasionar fallos difíciles de identificar.    Supongamos que tenemos la entidad *Application* en la que uno de sus atributos es de tipo *Status* (enumerado) y necesitamos una query que compruebe el *Status* de una aplicación.  Pregunta: ¿Qué tipo de dato le pasamos como parámetro? ¿Una cadena? ¿Un número?  Respuesta: El objeto de tipo enumerado. Esto quedaría de la siguiente forma:  @Query("select app from Application app where app.status = ?1")  Collection<Application> findApplicationsByStatus(Status status);  Consejo: si necesitamos comparar alguna fecha de la base de datos con el momento actual, en lugar de pasar el momento actual como parámetro en entrada, podemos utilizar en la propia query CURRENT\_TIMESTAMP. Por ejemplo:    @Query("select w from WarningNote w where w.moment < CURRENT\_TIMESTAMP  and w.trip.id = ?1")  Collection<WarningNote> findByPastMomentByTripId(int tripId); |

Ejemplo – Sintaxis de repositorios:

**@Repository**

public interface [Entidad]Repository extends JpaRepository

<[Entidad], Integer> {

**@Query(“[Consulta en JPQL]”)**

[Tipo] [nombreMétodo] ([Parámetros de entrada]);

// Así para cada consulta que se vaya a crear

}

# Anotaciones en servicios

Un **servicio (*services*)** es una clase que implementa la lógica de negocio y las reglas de negocio de una entidad concreta.

Para este tipo de clases debemos incluir las siguientes anotaciones:

|  |
| --- |
| Anotaciones en los servicios |
| @Services: es una especialización de la anotación @Component. Indica que el servicio es una clase/componente de Spring, permitiendo considerarlo como uno de sus objetos. Se añade en la cabecera de la clase. |
| @Transactional: indica que los métodos incluidos en el servicio se van a ejecutar en el contexto de una transacción. Se añade en la cabecera de la clase. |
| @Autowired: indica a Spring que inserte una instancia de la interfaz del repositorio o servicio sobre la que se incluye. Gracias a esto, no tenemos que crear una implementación para esta interfaz; es Spring el que lo crea siempre que sea necesario. |

Ejemplo – Sintaxis de servicios:

**@Services**

public services [Entidad]Services {

// Managed repository ------------------------------------

**@Autowired**

private [Entidad]Repostory [entidad]Repository;

// Supporting services -----------------------------------

**@Autowired**

private [Servicio]Service [servicio]Service;

**@Autowired**

private [Servicio]Service [servicio]Service;

// Y así con todos los servicios que necesitemos

…

}

# Anotaciones en los test

**Testear un repositorio o servicio** significa escribir test informales que permiten confirmar que todo funciona correctamente.

Para este tipo de clases debemos incluir las siguientes anotaciones:

|  |
| --- |
| Anotaciones en los test |
| @RunWith(SpringJUnit4ClassRunner.class): indica a JUnit que debe usar el lanzador de Spring para que Spring pueda instanciar las clases requeridas por su código. Se añade en la cabecera de la clase. Permite ejecutar el código de prueba sobre Spring. |
| @ContextConfiguration(locations = {…}): especifica la configuración Spring que debe ser usada para arrancar nuestros test. En este caso, incluimos los ficheros de configuración que contienen:   * datasource.xml: información sobre la configuración de la base de datos. * packages.xml: información sobre la localización de los paquetes en los cuales se encuentran los repositorios y servicios. * spring/junit.xml: si entramos en el fichero spring.xml podemos ver que contiene los ficheros datasource.xml y packages.xml.   Se añade en la cabecera de la clase.  Aclaración: aunque ambas formas sean equivalentes, el uso de esta anotación varía según el cuatrimestre:   * 1º cuatrimestre: se añaden datasource.xml y packages.xml.   @ContextConfiguration(locations =  {"classpath:spring/datasource.xml",  "classpath:spring/config/packages.xml"  })   * 2º cuatrimestre: se añade spring/junit.xml.   @ContextConfiguration(locations =  {“classpath:spring/junit.xml”  }) |
| @Transactional: indica que los métodos incluidos en el test se van a ejecutar en el contexto de una transacción. Se añade en la cabecera de la clase. Es necesario para que podamos deshacer el código ejecutado (rollback).  Nota del profesor: Si no se pone “a saber lo que hace Spring”. |
| @Autowired: indica a Spring que inserte una instancia de la interfaz del servicio sobre la que se incluye. Gracias a esto, no tenemos que crear una implementación para esta interfaz; es Spring el que lo crea siempre que sea necesario. |
| @Test (…): marca el método que va a testear un servicio. Si no se indica, el método no se ejecutará y por tanto no se testeará el servicio.  Aclaración: el uso de esta anotación varía según el cuatrimestre:   * 1º cuatrimestre: no insertamos ningún parámetro a la anotación @Test. * 2º cuatrimestre: dependiendo del tipo de test utilizaremos añadiremos parámetros o no:   Casos de prueba positivos: se centran en comprobar el comportamiento normal de la aplicación. En este caso, no añadimos ningún parámetro a la anotación:  @Test  Casos de prueba negativos: se centran en comprobar el comportamiento anómalo de la aplicación (“meter los dedos al sistema”). La idea es, definir como parámetro de entrada la excepción que se espera que el caso de prueba debe lanzar. Para ello, añadimos como parámetro a la anotación expected:  @Test (expected = [ExceptionEsperada].class) |
| @Before: el código especificado se ejecuta antes de ejecutar cada uno de los test. Utilizar está anotación implica utilizar el método setUp, el cual sólo debemos sobrescribir si realmente se necesita, pero por regla general no es necesario ya que Spring se encarga de configurar el fixture requerido para obtener los repositorios, servicios, controladores, convertidores y muchos componentes de trabajo.  @Before  void setUp();  Importante: para que funcione este método, la clase de test debe extender de AbstractTest. |
| @After: el código especificado se ejecuta despuésde ejecutar cada uno de los test. Utilizar está anotación implica utilizar el método tearDown, el cual sólo debemos sobrescribir si realmente se necesita, pero por regla general no es necesario ya que Spring se encarga de configurar el fixture requerido para obtener los repositorios, servicios, controladores, convertidores y muchos componentes de trabajo.  @After  void tearDown();  Importante: para que funcione este método, la clase de test debe extender de AbstractTest. |

Ejemplo – Sintaxis de test:

**@RunWith(SpringJUnit4ClassRunner.class)**

**@ContextConfiguration(locations =**

**{“classpath:spring/datasource.xml”,**

**“classpath:spring/packages.xml”})**

**@Transactional**

public class [Entidad]ServiceTest {

// Service under test ------------------------------------

**@Autowired**

private [Entidad]Service [entidad]Service;

**@Autowired**

private [Entidad]Service [entidad]Service;

// Así con todos los servicios que necesitemos para testear

// la entidad

// Test --------------------------------------------------

**@Test**

public void test[método][Entidad] () {

…

}

**@Test**

public void test[método][Entidad] () {

…

}

// Así hasta testear todos los métodos de los servicios.

…

}

# Anotaciones en los controladores

Un **controlador (*controller*)** es una clase cuyos objetos orquestan una serie de servicios para servir solicitudes HTTP a algunas URL.

Para este tipo de clases debemos incluir las siguientes anotaciones:

|  |
| --- |
| Anotaciones en los controladores |
| @Controller: es una especialización de la anotación @Component. Indica que el controlador es una clase/componente de Spring, permitiendo considerarlo como uno de sus objetos. Se añade en la cabecera de la clase. |
| @RequestMapping: instruye a Spring para redirigir cada petición recibida a un controlador determinado, es decir, permite asociar una URL a un controlador. Esta anotación se añade tanto la cabecera de la clase como en los métodos del controlador.  Explicación: para entender cómo funciona esta anotación debemos conocer cómo diseñamos las URL de nuestra aplicación. Estas siguen la siguiente sintaxis:  protocol://domain:port/app-context/entity/authorities/action.do?params   * app-context: se refiere al contexto de la aplicación, por ejemplo, el nombre de la aplicación (Acme-Certification). * entidad: se refiere al nombre de la entidad (Application, Warning-Notes,…) * authorities: se refiere al rol de nuestra aplicación (user, admin, manager). Este puede omitirse dependiendo de cada caso. * action.do?params: indica la acción que vamos a realizar (action) y los parámetros (params) necesarios para realizar esta acción si son necesarios.   Supongamos que necesitamos controlar la siguiente URL  [POR TERMINAR] |
| @RequestParam( required = false ) [TipoParámetro] parámetro: <http://www.jtech.ua.es/j2ee/publico/spring-2012-13/sesion03-apuntes.html> |
| @Valid: valida datos de forma recursiva. Normalmente se aplica a tipos de datos definidos por el usuario. También se utiliza en las restricciones de dominio.  Por regla general, utilizaremos esta etiqueta con aquellos tipos definidos por nosotros.  Ejemplo: podemos usar esta anotación para validar los parámetros de entrada en los métodos del controlador que verifique las restricciones de validación que definimos en las entidades del dominio.  El resultado de esta “pre-validación” es accesible a través del segundo parámetro de tipo *BindingResult*. Esta clase tiene métodos para averiguar qué errores se han producido e incluso añadir otros nuevos.  Este tipo de validaciones se realizan en los métodos del controlador para editar (*edit*) y guardar (*save*) un objeto de una entidad concreta.  Supongamos que tenemos un controlador en el que queremos guardar un “announcement”. Esta etiqueta validará todas aquellas restricciones definidas en la entidad Announcement del parámetro de entrada:  @RequestMapping(value=”/edit”, method=RequestMethod.POST, params=”save”)  public ModelAndView save(@Valid Announcement announcement,  BindingResult binding){  ModelAndView result = null;  if(binding.hasErrors(){  result = new ModelAndView(“redirect:list.do”);  } else {  try {  announcementService.save(announcement);  result = new ModelAndView(“redirect:list.do”);  } catch (Throwable oops) {  result = createEditModelAndView(announcement,  “announcement.commit.error”);  }  }  Error común: debemos tener cuidado con los atributos ocultos en las vistas.  Supongamos que necesitamos crear un objeto de la entidad *Announcement*. El atributo momentMade no puede ser nulo y debe ser definido por el sistema.   |  |  | | --- | --- | |  | El método *create* de *AnnouncementService* crea un nuevo objeto de tipo *Announcement*. La vista establece el atributo *momentMade* como hidden.  El usuario rellena el formulario y pulsa el botón de enviar, llegando al método *save* de la clase *AnnouncementController*. |   Problema: la anotación @Valid comprobará las restricciones del dominio de la entidad *Announcement* y dará un error porque el parámetro *momentMade* es nulo.  Solución: asignar un valor al atributo *momentMade* en el método *create* de *AnnouncementService*. Cuando llegue al método *save* de *AnnouncementController* cumplirá con todas las restricciones y lo guardará mediante el método *save* de *AnnouncementService*.  Como la fecha definida al principio no contiene el valor definido necesario, volvemos a definir el atributo *momentMade* dentro del método *save* y lo persistiremos correctamente. |
| @ModelAttribute: vincula un parámetro de entrada de un método del controlador a un atributo del modelo (*model attribute*) expuesto en la vista de nuestra aplicación web.  Error común: partimos de la entidad Message y queremos crear un nuevo *Message*. Supongamos que introducimos mal los datos y hacemos clic en Guardar. Como los datos se han introducido de forma errónea, el controlador correspondiente volverá al formulario mostrando los errores en los campos que corresponda.    Problema: los errores no se muestran en formulario porque el nombre de la entidad coincide con la variable message encargada de enviar los errores del formulario. Por tanto, debemos de ser capaces de distinguir el objeto de formulario que se está editando y los errores del proyecto.  Solución 1: cambiar el nombre de la entidad para evitar la confusión.  Solución 2: utilizar la anotación @ModelAttribute de la siguiente forma:   * edit.jsp de Message:   <form:form action="message/[rol]/notification.do"  modelAttribute="messageNotification">  …  </form>   * Message[rol]Controller.java de Message:   @RequestMapping(value = "/notification",  method = RequestMethod.POST,  params = "notify")  public ModelAndView notification(  @Valid @ModelAttribute("messageNotification") final Message message,  final BindingResult binding) {  …  } |
|  |