Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

Configuración y uso de Jackson

Fecha: 22/06/2012 | **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.ejie.es

Control de documentación

Título de documento: Configuración y uso de Jackson

Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	22/06/2012	Primera versión.

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

1 Introducción

Mediante el presente documento se pretende detallar el modo de uso y configuración del serializador/deserializador Jackson.

Jackson es una librería Java para el procesamiento de información en formato JSON.

2 Arquitectura

La capa de Control (Spring MVC) trabaja constantemente con el modelo para intercambiar información con la vista. Uno de los formatos utilizados por UDA para el intercambio de datos es el formato JSON y estas son las ventajas de su uso son:

- Formato ligero.
- Sintaxis de comprensión fácil para los humanos.
- Fácil de interpretar y generar programáticamente.

Para ampliar la información relacionada con JSON se puede acudir a la siguiente página http://www.json.org/

En el proceso de intercambio de información entre la capa de control y la vista se producen dos tipos de acciones:

- Serialización: Se trata del proceso de generar objetos JSON a partir de objetos Java. Este es el proceso que se realiza a la hora de enviar información desde la capa de control a la vista.
- Deserialización: Se trata del proceso de generar objetos Java a partir de objetos JSON. Este en el proceso que se realizar a la hora de enviar información desde la capara de la vista a la de control.

Para simplificar los procesos de serialización y deserialización de la información intercambiada, se recurre al uso de la librería Jackson. Esta librería proporciona una serie de funcionalidades para el procesamiento de información en formato JSON.

El uso de la librería Jackson se realiza mediante la integración que permite Spring MVC.

El uso de los *HttpMessageConverters* de Spring realiza la conversión de la información recibida por las *HTTP request* y enviadas por las *http responses* a objetos Java. Entre las implementaciones de *HttpMessageConverter* que proporciona Spring se encuentra la clase *MappingJacksonHttpMessageConverter.* Esta implementación proporciona la habilidad de leer y escribir objetos JSON mediante el uso de la clase *ObjectMapper* de Jackson. El mapeo JSON puede ser configurado mediante los mecanismos que ofrece la librería de Jackson.

Para ampliar las funcionalidades y el nivel de configuración del componente, se ha implementado un *HttpMessageConverter* propio, *UdaMappingJacksonHttpMessageConverter,* el cual extiende del proporcionado por Spring.

En los siguientes apartados se detallarán las mejoras introducidas. Una breve enumeración de las mismas es la siguiente:

- Configuración declarativa de serializadores y deserializadores.
- Facilitar la configuración del comportamiento de serialización y deserialización de Jackson.
- Permitir la serialización selectiva de una serie de propiedades de un bean dependiendo de información introducida en las cabeceras de la petición HTTP.
- Permitir la deserialización de múltiples entidades enviadas en una misma petición. Esta funcionalidad se activa por defecto simplemente con configurar el UdaModule

2.1 Dependencias

La librería de Jackson no tiene dependencias de ninguna otra librería externa. Únicamente se deberá de incluir en la aplicación la librería de Jackson y la librería x38.

Para poder hacer uso de la librería en la aplicación se deberá de 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.ejie.x38.version>2.0.0</com.ejie.x38.version>
    </properties>
    <dependencies>
        ...
        ...
        <!-- Jackson JSON Mapper -->
        <dependency>
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>1.9.7</version>
        </dependency>
        ...
        ...
        <dependency>
            <groupId>com.ejie.x38</groupId>
            <artifactId>x38ShLibClasses</artifactId>
            <version>${com.ejie.x38.version}</version>
        </dependency>
        ...
        ...
    <dependencies>
</project>
```

2.2 Clases Java

Para facilitar el uso y configuración de Jackson, así como para aportar nuevas funcionalidades, se proporcionan una serie de clases Java incluidas en la librería x38:

- com.ejie.x38.serialization.CustomSerializer
    - Implementación de un serializador propio que permite la serialización selectiva de unas propiedades determinadas de un bean.
- com.ejie.x38.serialization.JsonDateDeserializer
    - Clase que permite la deserialización de fechas.
- com.ejie.x38.serialization.JsonDateSerializer
    - Clase que permite la serialización de fechas.
- com.ejie.x38.serialization.JsonDateTimeDeserializer
    - Clase que permite la deserialización de fechas incluyendo en el proceso la hora.

- com.ejie.x38.serialization.JsonDateTimeSerializer
  - Clase que permite la serialización de fechas incluyendo en el proceso la hora.
- com.ejie.x38.serialization.JsonNumberDeserializer
  - Clase que permite la deserialización de objetos de tipo `java.math.BigDecimal` dependiendo del formato especificado por la Locale.
- com.ejie.x38.serialization.JsonNumberSerializer
  - Clase que permite la serialización de objetos de tipo `java.math.BigDecimal` dependiendo del formato especificado por la Locale.
- com.ejie.x38.serialization.JsonTimeDeserializer
  - Clase que permite la deserialización de fechas teniendo únicamente en cuenta la hora.
- com.ejie.x38.serialization.JsonTimeSerializer
  - Clase que permite la serialización de fechas teniendo únicamente en cuenta la hora.
- com.ejie.x38.serialization.MultiModelDeserializer
  - Clase que permite la serialización de varias entidades en la misma petición.
- com.ejie.x38.serialization.ThreadSafeCache
  - Clase de utilidad que permite almacenar propiedades de modo seguro en el Thread local para permitir la transferencia de información entre los componentes de UDA que utilizan Jackson.
- com.ejie.x38.serialization.UdaMappingJacksonHttpMessageConverter
  - Implementación del *HttpMessageConverter* propio para la gestión de JSON. Esta clase extiende del *MappingJacksonHttpMessageConverter* proporcionado por Spring MVC para la integración con Jackson.
- com.ejie.x38.serialization.UdaModule
  - Clase de UDA que se encarga de implementar el módulo de Jackson con la configuración necesaria para dotarle de las funcionalidades requeridas.

3 Configuración

La configuración de Jackson se realiza a nivel de WAR. Cada aplicación deberá realizar tantas configuraciones como módulos web tenga.

El código necesario para la configuración de Jackson se realiza en el fichero *jackson-config.xml*. Para que Spring acceda a la configuración definida en dicho fichero se deberán inlcuir en el fichero *app-config.xml* la sentencia correspondiente a la carga del mismo:

```
<import resource="jackson-config.xml" />
```

A continuación se detallará el código incluido en el fichero de configuración de Jackson.

El primer paso es definir el serializador *CustomSerializer* que se incluye en x38. Este serializador permite realizar la serialización de un bean únicamente teniendo en cuenta unas propiedades determinadas.

```
<bean id="customSerializer" class="com.ejie.x38.serialization.CustomSerializer" />
```

Una de las nuevas inclusiones en la configuración del componente Jackson es el uso de módulos. Estos permiten extender y configurar las capacidades y funcionalidades de procesamiento de objetos JSON de Jackson. En el siguiente apartado se explicará de manera mas detallada cada una de las propiedades de configuración del mismo.

```
<bean id="udaModule" class="com.ejie.x38.serialization.UdaModule" >
    <property name="serializers">
       <util:map>
           <entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="customSerializer" />
          <entry key="#{T(com.ejie.x21a.model.Comarca)}" value-ref="customSerializer" />
          <entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="customSerializer" />
          <entry key="#{T(com.ejie.x21a.model.DepartamentoProvincia)}" value-ref="customSerializer"/>
          <entry key="#{T(com.ejie.x21a.model.Localidad)}" value-ref="customSerializer" />
          <entry key="#{T(com.ejie.x21a.model.Provincia)}" value-ref="customSerializer" />
          <entry key="#{T(com.ejie.x21a.model.Usuario)}" value-ref="customSerializer" />
       </util:map>
    </property>
    <property name="serializationInclusions" ref="serializationInclusions" />
    <property name="serializationConfigFeatures" ref="serializationConfigFeatures" />
    <property name="deserializationConfigFeatures" ref="deserializationConfigFeatures" />
</bean>
```

El siguiente código determina los MediaTypes que Jackson va a soportar a la hora de procesar los JSON. Por defecto Jackson está configurado para procesar las peticiones cuyos datos sean de tipo application/json. Para

poder utilizar el componente RUP upload desde un navegador IE8 es necesario indicarle que procese también los de tipo text/plain.

```
<util:list id="jacksonSupportedMediaTypes">
	<bean class="org.springframework.http.MediaType">
		<constructor-arg value="text" />
		<constructor-arg value="plain" />
		<constructor-arg
			value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConve
			rter).DEFAULT_CHARSET}" />
	</bean>
	<bean class="org.springframework.http.MediaType">
		<constructor-arg value="application" />
		<constructor-arg value="json" />
		<constructor-arg
			value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConve
			rter).DEFAULT_CHARSET}" />
	</bean>
</util:list>
```

Por último se deberá de declarar el message converter para Jackson implementado en x38. Esta clase extiende la clase *MappingJacksonHttpMessageConverter* para permitir realizar la configuración de muchas de las funcionalidades que se exponen en el presente documento.

```
<bean id="udaMappingJacksonHttpMessageConverter"
	class="com.ejie.x38.serialization.UdaMappingJacksonHttpMessageConverter">
	<property name="supportedMediaTypes" ref="jacksonSupportedMediaTypes" />
	<property name="udaModule" ref="udaModule" />
</bean>
```

Una vez finalizada la configuración en el fichero *jackson-config.xml* se deberá de indicar en el fichero *mvc-config.xml* el uso de *udaMappingJacksonHttpMessageConverter* por los message converter de spring.

```
<bean id="requestMappingHandlerAdapter"
	class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

	<property name="messageConverters">
	    <list>
	       <ref bean="udaMappingJacksonHttpMessageConverter"/>
	    </list>
	</property>
</bean>
```

4 UdaModule

Los módulos de Jackson son una nueva funcionalidad que se introdujo a partir de la versión 1.7. Mediante los módulos se permite extender la funcionalidad proporcionando para tal fin un modelo potente y flexible.

El módulo de UDA implementado proporciona las siguientes funcionalidades:

- Configuración declarativa de serializadores y deserializadores.
- Facilitar la configuración del comportamiento de serialización y deserialización de Jackson.
- Permitir la serialización selectiva de una serie de propiedades de un bean dependiendo de información introducida en las cabeceras de la petición HTTP.
- Permitir la deserialización de múltiples entidades enviadas en una misma petición. Esta funcionalidad se activa por defecto simplemente con configurar el UdaModule

4.1 Configuración

Tal y como se ha comentado en el apartado anterior, vamos a proceder a detallar la configuración y uso del UdaModule. Para ello vamos a partir de la siguiente configuración básica:

```
<bean id="udaModule" class="com.ejie.x38.serialization.UdaModule" >
	<property name="serializers">
		<!-- Mapa en el que se indican los serializadores-->
	</property>
	<property name="deserializers">
		<!-- Mapa en el que se indican los deserializadores-->
	</property>
	<property name="serializationInclusions">
		<!-- Lista de propiedades de configuración -->
	</property>
	<property name="serializationConfigFeatures">
		<!-- Mapa de propiedades de configuración de serialización -->
	</property>
	<property name="deserializationConfigFeatures">
		<!-- Mapa de propiedades de configuración de deserialización -->
	</property>
</bean>
```

Existen cinco propiedades principales mediante las cuales se puede configurar las características y funcionalidades de Jackson. Estas propiedades son las siguientes:

- **serializers**: Esta propiedad acepta un mapa mediante el cual se puede especificar un serializador concreto para un bean determinado. Un ejemplo de configuración de esta propiedad podría ser la siguiente:

```
<property name="serializers">
	<util:map>
		<entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="customSerializer" />
		<entry key="#{T(com.ejie.x21a.model.Comarca)}"value-ref="customSerializer" />
		<entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="customSerializer"
		/>
	</util:map>
</property>
```

En este caso estamos indicando que los beans *Alumno, Comarca* y *Departamento* pueden hacer uso del serializador de UDA *customSerializer*. En el *apartado 4.2* se explicará la finalidad y el modo de uso de este serializador.

- **deserializers**: Esta propiedad acepta un mapa mediante el cual se puede especificar un deserializador concreto para un bean determinado. Un ejemplo de configuración de esta propiedad podría ser la siguiente:

```
<property name="deserializers">
    <util:map>
        <entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="alumnoSerializer" />
        <entry key="#{T(com.ejie.x21a.model.Comarca)}" value-ref="comarcamSerializer " />
        <entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="departamentoSerializer"
        />
    </util:map>
</property>
```

En el ejemplo se está indicando que para los beans *Alumno, Comarca* y *Departamento* se va a utilizar un deserializador concreto para cada uno de ellos.

Al contrario que ocurre con los serializadores, desde UDA no se configura de manera explícita ningún deserializador. Los que se definan deberán ser implementados por las aplicaciones para cubrir una necesidad concreta.

- **serializationInclusions**: Mediante esta propiedad se puede configurar las propiedades que van a ser incluidas en la serialización. Los valores posibles son:
    - **ALWAYS**: Las propiedades van a ser siempre incluidas en la serialización, independientemente de su valor.
    - **NON_DEFAULT**: Solo se incluirán en la serialización aquellas propiedades cuyo valor difiera de los valores por defecto (valores con los que se inicializan al instanciar el bean mediante el constructor sin argumentos).
    - **NON_EMPTY**: Solo se incluirán en la serialización las propiedades cuyos valores no sean *null* o vacíos.
    - **NON_NULL**: Únicamente se incluirán en la serialización aquellas propiedades que no sean nulas.

```
<property name="serializationInclusions">
    <util:list>
        <value type="org.codehaus.jackson.map.annotate.JsonSerialize.Inclusion">
            NON_NULL
        </value>
    </util:list>
</property>
```

- **serializationConfigFeatures**: Esta propiedad acepta un mapa para configurar el proceso de serialización. Cada entrada del mapa consta de una clave (*key*) que indica la característica a configurar y del valor del elemento (*value*) que puede ser *true* o *false.*

```
<property name="serializationConfigFeatures">
    <util:map>
        <entry
        key="#{T(org.codehaus.jackson.map.SerializationConfig$Feature).SORT_PROPERTIES_ALPHABET
        ICALLY}" value="true" />
        <entry
        key="#{T(org.codehaus.jackson.map.SerializationConfig$Feature).FAIL_ON_EMPTY_BEANS}"
```

```
        value="true" />
    </util:map>
</property>
```

En el ejemplo se configuran las propiedades SORT_PROPERTIES_ALPHABETICALLY y FAIL_ON_EMPTY_BEANS.

La lista completa de posibles características configurables para el proceso de serialización se muestra en la web del proyecto de Jackson: http://wiki.fasterxml.com/JacksonFeaturesSerialization

- **deserializationConfigFeatures**: Esta propiedad acepta un mapa para configurar el proceso de deserialización. Cada entrada del mapa consta de una clave (*key*) que indica la característica a configurar y del valor del elemento (*value*) que puede ser *true* o *false*

```
<property name="deserializationConfigFeatures">
    <util:map>
        <entry
        key="#{T(org.codehaus.jackson.map.DeserializationConfig$Feature).FAIL_ON_UNKNOWN_PROPER
        TIES}" value="true" />
        <entry
        key="#{T(org.codehaus.jackson.map.DeserializationConfig$Feature).ACCEPT_EMPTY_STRING_AS
        _NULL_OBJECT}" value="true" />
    </util:map>
</property>
```

En el ejemplo se configuran las propiedades FAIL_ON_UNKNOWN_PROPERTIES y ACCEPT_EMPTY_STRING_AS_NULL_OBJECT.

La lista completa de posibles características configurables para el proceso de deserialización se muestra en la web del proyecto de Jackson: http://wiki.fasterxml.com/JacksonFeaturesDeserialization

4.2 Serialización selectiva

Mediante el uso del *customSerializer* se permite realizar una serialización selectiva de las propiedades de un bean. La finalidad de este serializador es facilitar el uso de componentes que necesitan de una lista de resultados formados por elementos *clave-valor* (ej. combo, autocomplete).

De este modo basta con indicarle al serializador que propiedades del bean van a ser la utilizadas para obtener la clave y el valor. El uso o no del serializador se determina mediante una cabecera en la petición HTTP que contendrá la correlación entre las propiedades y la estructura devuelta.

La gestión de esta cabecera se realiza internamente por los componentes *rup.combo* y *rup.autocomplete*, por lo que el desarrollador no debe preocuparse por el envío de la cabecera. En caso de querer usarlo desde un desarrollo propio se debería de generar de manera manual.

Así pues este serían los pasos a seguir para utilizar este modo de serialización:

1. Tal y como se indica en el apartado anterior, se deberán configurar los beans que van a hacer uso del serializador. Para que un bean sea procesado por el *customSerializer* deberá de ser especificado en la siguiente lista utilizada para inicializar la propiedad *serializers* del *UdaModule*:

```
<property name="serializers">
        <util:map>
                <entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="customSerializer" />
```

```
                <entry key="#{T(com.ejie.x21a.model.Comarca)}"value-ref="customSerializer" />
                <entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="customSerializer"
                />
        </util:map>
</property>
```

2. Supongamos que disponemos del un bean *Departamento:*

```
public class Departamento  implements java.io.Serializable {

        private BidDecimal id;
        private String descEs;
        private String descEu;

        // Getters y setters de las propiedades
}
```

El objetivo es devolver una lista de departamentos cuya propiedad *id* sea el *value* y la propiedad *descEs* sea el *label*. Para ello se envía la siguiente información en la cabecera de la *request*:

```
RUP     {"label":"descEs","value":"id"}
```

3. El método del controller correspondiente procesaría la petición y devolvería la lista de elementos recuperados. El método anotado mediante *@ResponseBody* utilizaría el *messageConverter* de jackson:

```
@RequestMapping(method = RequestMethod.GET)
public @ResponseBody List<Departamento> find(Object parametros) {

        // Obtención de la lista de resultados

        return listaDepartamentos;

}
```

El serializador realizará el mapeo entre el valor que contienen las propiedades y el objeto JSON resultante que sería el siguiente:

```
[
        {"value":"1","label":"Departamento 1"},
        {"value":"2","label":"Departamento 2"}
        {"value":"3","label":"Departamento 3"}
]
```

4.3 Deserialización de multientidades

El uso de UdaModule proporciona a Jackson la capacidad de deserializar varias entidades diferentes no anidadas enviadas en la misma petición.

Por ejemplo, supongamos que se debe de enviar desde la capa de presentación cierta información introducida por el usuario. El *databinding* de estos datos debe de ser realizado sobre dos beans entre los cuales no existe anidamiento:

```
public class Alumno  implements java.io.Serializable {

        private BidDecimal id;
```

```
        private String nombre;
        private String apellido1;
        private String apellido2;

        // Getters y setters de las propiedades
}

public class Departamento  implements java.io.Serializable {

        private BidDecimal id;
        private String descEs;
        private String descEu;

        // Getters y setters de las propiedades
}
```

El problema de esta situación radica en el proceso de deserialización del JSON enviado y a partir del cual se debe realizar el *databinding* sobre el parámetro del método del controller correspondiente que se ha anotado con `@RequestBody.`

Una solución sería la creación de un nuevo bean que contenga ambos beans como propiedades:

```
public class BeanAuxiliar  implements java.io.Serializable {

        private Alumno alumno;
        private Departamento departamento;

        // Getters y setters de las propiedades
}
```

De este modo, este sería un ejemplo del método del controller que deberíamos de implementar (ej. para el método *add*):

```
@RequestMapping(method = RequestMethod.POST)
public @ResponseBody Usuario add(@RequestBody BeanAuxiliar beanAuxiliar) {

        // Obtención del alumno
        beanAuxiliar.getAlumno();
        // Obtención del departamento
        beanAuxiliar.getDepartamento();

}
```

Para evitar la creación de un nuevo tipo de bean cada vez que se da esta necesidad se ha implementado una nueva funcionalidad de Jackson. Esta funcionalidad es proporcionada por el UdaModule y permite realizar el *databinding* de varias entidades sobre un parámetro del método de tipo *HashMap.*

Para poder determinar en cada petición si se va a utilizar o no el *databinding* múltiple, se incluye una cabecera en la petición, RUP_MULTI_ENTITY. En caso de ser *true* se activará el comportamiento especial.
Este sería el proceso completo de *databinding* múltiple:

1. Partimos de un formulario definido en la JSP en el que se dispone de varios campos correspondientes a varias entidades:

```
<form id="formMultientidades" action='/multientidades' method="post">
       <input type="text" name="alumno.id" />
       <input type="text" name="alumno.nombre" />
```

```
        <input type="text" name="alumno.apellido1" />
        <input type="text" name="alumno.apellido2" />
        <input type="text" name="departamento.id" />
        <input type="text" name="departamento.descEs" />
        <input type="text" name="departamento.descEu" />
        <input type="submit" value="Enviar" />
</form>
```

2. El usuario rellena los datos y estos se envían en formato `application/json; charset=UTF-8` junto con la cabecera RUP_MULTI_ENTITY `true`.

```
Content-Type          application/json; charset=UTF-8
RUP_MULTI_ENTITY      true

{
        "alumno":
                {"id":"1","nombre":"Nom","apellido1":"ap1","apellido2":"ap2"},
        "departamento":
                {"id":"3","descEs":"Dep_es","descEu":"Dep_eu"},

        "rupEntityMapping":{
                "alumno":"com.ejie.x21a.model.Alumno",
                "departamento":"com.ejie.x21a.model.Departamento"
        }
}
```

En el objeto JSON se envía la siguiente información:

- Los datos correspondientes a cada una de las entidades sobre las que se debe realizar el *databinding* en formato JSON.
- Un objeto de nombre *rupEntityMapping* que contiene la correlación entre los nombres de los objetos y el tipo Java.

3. La petición se mapea sobre el método del controller correspondiente. El tipo de parámetro del parámetro sobre el que se van a crear las nuevas entidades debe ser de tipo *java.util.Map*:

```
@RequestMapping(value = "form/multientidades", method = RequestMethod.POST)
public @ResponseBody Object getFormmMultientidades(@RequestBody Map<String, Object>
        multiModelMap) {

        // Obtención del alumno
        Alumno alumno = (Alumno)multiModelMap.get("alumno");

        // Obtención del departamento
        Departamento departamento =
        (Departamento)multiModelMap.get("departamento");
}
```

El key de cada uno de los elementos del mapa y el tipo de objeto que se crea en dicho elemento se corresponde con la configuración indicada en el objeto JSON *rupEntityMapping.*

5 Anotaciones

Jackson permite la permite la personalización y configuración del proceso de serialización y deserialización mediante la inclusión de anotaciones en las propiedades y métodos de los beans que van a ser procesados. En este anexo vamos a centrarnos únicamente en explicar el uso de aquellas que se utilizan en el código generado por el plugin generador de código.

No obstante siempre se puede acudir a la documentación oficial de Jackson para profundizar en este y otros temas.

5.1 @JsonIgnore

Mediante esta anotación se puede determinar que una propiedad determinada no sea tenida en cuenta en el proceso de serialización/deserialización.

Un ejemplo puede ser una propiedad de un bean en la que se recupera el contenido de un campo BLOB de la base de datos, debido a que no se desea serializar en formato JSON un fichero completo.

NOTA: En versiones posteriores a Jackson 1.9, la anotación @JsonIgnore podía ser utilizada a nivel de método (p.e. un método getter de una propiedad). A partir de la versión 1.9 es posible anotar un método pero se toma como una anotación de propiedad. Es decir si estamos anotando el método getter mediante @JsonIgnore, esta propiedad no solo se ignorará en el proceso de serialización sino que también será obviada al realizar la deserialización.

Debido a esto es necesario utilizar la anotación @JsonProperty("propiedad") en el método que queremos que conserve su funcionalidad.

En el siguiente ejemplo realizaremos las anotaciónes correspondientes para que la clave de acceso de un alumno no sea procesada durante la serialización pero si en la deserialización.

```
public class Alumno  implements java.io.Serializable {

	private BidDecimal id;
	private String nombre;
	private String apellido1;
	private String apellido2;
	private String password;

	// Getters y setters de las propiedades

	@JsonIgnore
	public String getPassword() {
		return this.password;
	}

	@JsonProperty("password")
	public void setPassword(String password) {
		this.password = password;
	}
}
```

5.2 @JsonSerialize

Mediante esta anotación se permite especificar un serializador concreto que se va a aplicar a la propiedad. En la librería x38 se incluyen una serie de serializadores implementados por defecto:

- *com.ejie.x38.serialization.JsonDateSerializer*: Permite la serialización de fechas utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de serialización únicamente tiene en cuenta la parte de la fecha obviando la hora.
- *com.ejie.x38.serialization.JsonDateTimeSerializer*: Permite la serialización de fechas utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de serialización tiene en cuenta tanto la fecha como la hora.
- *com.ejie.x38.serialization.JsonNumberSerializer*: Clase que permite la serialización de objetos de tipo `java.math.BigDecimal` dependiendo del formato especificado por la Locale.
- com.ejie.x38.serialization.JsonTimeSerializer: Permite la serialización de fechas utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de serialización tiene en cuenta únicamente la hora desechando la parte de la fecha.

La anotación puede indicarse tanto a nivel de método (ej. getter) como a nivel de propiedad. Un ejemplo de uso para realizar la serialización de una propiedad de tipo *java.util.Date* sería el siguiente:

```
public class Alumno  implements java.io.Serializable {

	private Date fechaNacimiento;

	// Resto de propiedades y métodos


	@JsonSerialize(using = JsonDateSerializer.class)
	public Date getFechaNacimiento() {
		return this.fechaNacimiento;
	}
}
```

Del mismo modo que se ha creado este conjunto de serializadores, es posible la implementación propia de serializadores extra para facilitar el desarrollo de las aplicaciones. Para ello se deberá de implementar una nueva clase que extienda de `JsonSerializer<T>`

```
@Component
public class CustomSerializer extends JsonSerializer<CustomBean>{

	@Override
	public void serialize(CustomBean customBean, JsonGenerator gen, SerializerProvider provider)
			throws IOException, JsonProcessingException {

		StringBuffer stringBuffer = new StringBuffer();

		// Generar el string a mostrar en la serialización a partir del bean

		gen.writeString(stringBuffer.toString());
	}
}
```

5.3 @JsonDeserialize

Mediante esta anotación se permite especificar un deserializador concreto que se va a aplicar a la propiedad. En la librería x38 se incluyen una serie de deserializadores implementados por defecto:

- *com.ejie.x38.serialization.JsonDateDeserializer*: Permite la deserialización de fechas a partir de un *string* utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de deserialización únicamente tiene en cuenta la parte de la fecha obviando la hora.
- *com.ejie.x38.serialization.JsonDateTimeDeserializer*: Permite la deserialización de fechas a partir de un *string* utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de deserialización tiene en cuenta tanto la fecha como la hora.
- *com.ejie.x38.serialization.JsonNumberDeserializer*: Clase que permite la deserialización de objetos de tipo `java.math.BigDecimal` a partir de un *string* dependiendo del formato especificado por la Locale.
- com.ejie.x38.serialization.JsonTimeDeserializer: Permite la deserialización de fechas a partir de un *string* utilizando para ello el formato correspondiente al idioma utilizado por la aplicación. El formato de deserialización tiene en cuenta únicamente la hora desechando la parte de la fecha.

La anotación puede indicarse tanto a nivel de método (ej. setter) como a nivel de propiedad. Un ejemplo de uso para realizar la deserialización de una propiedad de tipo *java.util.Date* sería el siguiente:

```
public class Alumno  implements java.io.Serializable {

	private Date fechaNacimiento;

	// Resto de propiedades y métodos

	@JsonDeserialize(using = JsonDateDeserializer.class)
	public void setFechaNacimiento(Date fechaNacimiento) {
		this.fechaNacimiento = fechaNacimiento;
	}
}
```

Del mismo modo que se ha creado este conjunto de deserializadores, es posible la implementación propia de deserializadores extra para facilitar el desarrollo de las aplicaciones. Para ello se deberá de implementar una nueva clase que extienda de `JsonDeserializer<T>`

```
@Component
public class JsonCustomDeserializer extends JsonDeserializer<CustomBean> {

	@Override
	public CustomBean deserialize(JsonParser par, DeserializationContext ctx)
			throws IOException, JsonProcessingException {

		CustomBean customBean = new CustomBean();
		String dateText = par.getText();

		// Generar el bean a partir del sptring que se debe deserializar

		return customBean;
	}
}
```

6 Migración desde versiones previas a UDA v2.0.0

Haciendo uso de las nuevas clases incluidas en Spring para la interacción con Jackson, se han definido una serie de mejoras que se traducen en una configuración más flexible y potente del mismo.

Del mismo modo se han incluido mejoras en el proceso de serialización/deserialización permitiendo el tratamiento de múltiples entidades en la misma petición, así como de un soporte integro con entidades anidadas mediante el uso de la notación *dot*.

En este apartado se van a detallar los cambios introducidos en el componente Jackson implementado para la versión v2.0.0 de UDA. Del mismo modo se detallará el proceso que deberá de seguir una aplicación para adaptar su código, generado para una versión anterior de UDA, a la nueva configuración e implementación.

A continuación se detalla la arquitectura de las nuevas clases y su configuración.

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.ejie.x38.control.JsonDateDeserializer
- com.ejie.x38.control.JsonDateSerializer
- com.ejie.x38.control.JsonDateTimeDeserializer
- com.ejie.x38.control.JsonDateTimeSerializer
- com.ejie.x38.control.JsonTimeDeserializer
- com.ejie.x38.control.JsonTimeSerializer
- com.ejie.x38.serialization.CustomObjectMapper
- com.ejie.x38.serialization.CustomSerializerFactoryRegistry

6.2 *Nuevas clases*

A continuación se detallan las nuevas clases incluidas en x38:

- com.ejie.x38.serialization.JsonDateDeserializer
  - Clase que permite la deserialización de fechas.
- com.ejie.x38.serialization.JsonDateSerializer
  - Clase que permite la serialización de fechas.
- com.ejie.x38.serialization.JsonDateTimeDeserializer
  - Clase que permite la deserialización de fechas incluyendo en el proceso la hora.
- com.ejie.x38.serialization.JsonDateTimeSerializer
  - Clase que permite la serialización de fechas incluyendo en el proceso la hora.
- com.ejie.x38.serialization.JsonNumberDeserializer
  - Clase que permite la deserialización de objetos de tipo *java.math.BigDecimal* dependiendo del formato especificado por la *Locale*.
- com.ejie.x38.serialization.JsonNumberSerializer

  - Clase que permite la serialización de objetos de tipo *java.math.BigDecimal* dependiendo del formato especificado por la *Locale*.

- com.ejie.x38.serialization.JsonTimeDeserializer
  - Clase que permite la deserialización de fechas teniendo únicamente en cuenta la hora.
- com.ejie.x38.serialization.JsonTimeSerializer
  - Clase que permite la serialización de fechas teniendo únicamente en cuenta la hora.
- com.ejie.x38.serialization.MultiModelDeserializer
  - Clase que permite la serialización de varias entidades en la misma petición.
- com.ejie.x38.serialization.UdaModule
  - Clase de UDA que se encarga de implementar el módulo de Jackson con la configuración necesaria para dotarle de las funcionalidades requeridas.

6.3 Migración de código

Como ya se ha comentado anteriormente, la nueva versión de Jackson que UDA v2.0.0 trae incorporada es en su mayor medida retrocompatible. El único cambio obligatorio que se ha de realizar por parte de las aplicaciones que deseen actualizar a la versión v2.0.0 es lo referente a la anotación @JsonIgnore (Apartado 5.1. @JsonIgnore).

Únicamente con el ajuste que se indica, la implementación existente de una aplicación que hace uso de una versión anterior de UDA, es perfectamente válida y funcional. El único inconveniente es que sin adoptar la configuración que se indica en el presente documento no se va a poder hacer uso de las nuevas funcionalidades incorporadas.

En este apartado se van a exponer una serie de pasos a seguir para facilitar el proceso de migración a la nueva configuración y así tratar de lograr que el máximo número de aplicaciones se puedan beneficiar de las nuevas funcionalidades.

Debido a que la configuración de Jackson se realiza a nivel de WAR, cada aplicación deberá realizar estas modificaciones tantas veces como módulos web tenga.

Los pasos a seguir son:

1. La configuración de Jackson se ha separado del fichero *spring-mvc.xml*. Se deberá crear un nuevo fichero de configuración de Spring, *jackson-config.xml*, situado en *<xxxYYY>War/WebContent/WEB-INF/spring/.*

   El contenido inicial del fichero será el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
              http://www.springframework.org/schema/mvc
              http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
              http://www.springframework.org/schema/context
              http://www.springframework.org/schema/context/spring-context-3.1.xsd
              http://www.springframework.org/schema/util
              http://www.springframework.org/schema/util/spring-util-3.1.xsd">
```

```
</beans>
```

2. Se deberá modificar el fichero *<xxxYYY>War/WebContent/WEB-INF/spring/app-config.xml* para añadir la referencia al nuevo fichero y que sea procesado al inicio. Se deberá añadir la entrada resaltada:

```
<import resource="jackson-config.xml" />
<import resource="mvc-config.xml" />
<import resource="log-config.xml"/>
<import resource="validation-config.xml"/>
<import resource="security-core-config.xml"/>
<import resource="security-config.xml"/>
```

3. Se deberá de modificar el fichero *<xxxYYY>War/WebContent/WEB-INF/spring/mvc-config.xml* para **eliminar** la siguiente configuración relacionada con Jackson:

```
...
...
<bean id="jacksonJsonObjectMapper" class="com.ejie.x38.serialization.CustomObjectMapper">
       <property name="customSerializerFactory" ref="jacksonJsonCustomSerializerFactory" />
</bean>
...
...
<bean id="customSerializer" class="com.ejie.x38.serialization.CustomSerializer" />

...
...

<bean class="com.ejie.x38.serialization.UdaMappingJacksonHttpMessageConverter">
       <property name="jacksonJsonObjectMapper" ref="jacksonJsonObjectMapper" />
       <property name="supportedMediaTypes">
              <list>
                     <bean class="org.springframework.http.MediaType">
                            <constructor-arg value="text" />
                            <constructor-arg value="plain" />
                            <constructor-arg
value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConverter).DEFAULT_CHARSET
}"/>
                     </bean>
                     <bean class="org.springframework.http.MediaType">
                            <constructor-arg value="application" />
                            <constructor-arg value="json" />
                             <constructor-arg
value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConverter).DEFAULT_CHARSET
}"/>
                     </bean>
              </list>
       </property>
</bean>
...
...
<bean id="jacksonJsonCustomSerializerFactory"
       class="com.ejie.x38.serialization.CustomSerializerFactoryRegistry">
        <property name="serializers">
            <map>
                <entry key="com.ejie.x21a.model.Alumno" value-ref="customSerializer" />
                <entry key="com.ejie.x21a.model.Comarca" value-ref="customSerializer" />
                <entry key="com.ejie.x21a.model.Departamento" value-ref="customSerializer" />
                <entry key="com.ejie.x21a.model.Usuario" value-ref="customSerializer" />
            </map>
        </property>
</bean>
...
...
```

4. En el fichero *<xxxYYY>War/WebContent/WEB-INF/spring/jackson-config.xml* se añadirá la nueva configuración de Jackson. Esta configuración es la descrita en el *apartado 3*:

```
<!-- Serializador utilizado por UDA para serializar unicamente determinadas propiedades  -->
<bean id="customSerializer" class="com.ejie.x38.serialization.CustomSerializer" />

<bean id="udaMappingJacksonHttpMessageConverter"
       class="com.ejie.x38.serialization.UdaMappingJacksonHttpMessageConverter">
       <property name="supportedMediaTypes" ref="jacksonSupportedMediaTypes" />
       <property name="udaModule" ref="udaModule" />
</bean>

<!-- Modulo de UDA para Jackson -->
<bean id="udaModule" class="com.ejie.x38.serialization.UdaModule" >
    <property name="serializers">
       <util:map>
            <entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="customSerializer" />
            <entry key="#{T(com.ejie.x21a.model.Comarca)}" value-ref="customSerializer" />
            <entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="customSerializer" />
            <entry key="#{T(com.ejie.x21a.model.Usuario)}" value-ref="customSerializer" />
       </util:map>
    </property>
</bean>

<!-- MediaTypes soportados por jackson -->
<util:list id="jacksonSupportedMediaTypes">
    <bean class="org.springframework.http.MediaType">
       <constructor-arg value="text" />
       <constructor-arg value="plain" />
       <constructor-arg
                value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConverte
                r).DEFAULT_CHARSET}" />
    </bean>
    <bean class="org.springframework.http.MediaType">
       <constructor-arg value="application" />
       <constructor-arg value="json" />
       <constructor-arg
                value="#{T(org.springframework.http.converter.json.MappingJacksonHttpMessageConverte
                r).DEFAULT_CHARSET}" />
    </bean>
</util:list>
```

Es importante prestar atención a la lista de serializadores indicados en la configuración de jackson. Para que la configuración sea equivalente deben de trasladarse todos los serializadores definidos en la configuración previa. Esta sería la transformación que se debe de realizar sobre ellos:

[Configuración existente]

```
<property name="serializers">
    <map>
       <entry key="com.ejie.x21a.model.Alumno" value-ref="customSerializer" />
       <entry key="com.ejie.x21a.model.Comarca" value-ref="customSerializer" />
       <entry key="com.ejie.x21a.model.Departamento" value-ref="customSerializer" />
       <entry key="com.ejie.x21a.model.Usuario" value-ref="customSerializer" />
    </map>
</property>
```

[Nueva configuración]

```
<property name="serializers">
    <util:map>
       <entry key="#{T(com.ejie.x21a.model.Alumno)}" value-ref="customSerializer" />
       <entry key="#{T(com.ejie.x21a.model.Comarca)}" value-ref="customSerializer" />
       <entry key="#{T(com.ejie.x21a.model.Departamento)}" value-ref="customSerializer" />
       <entry key="#{T(com.ejie.x21a.model.Usuario)}" value-ref="customSerializer" />
    </util:map>
</property>
```

5. Por último se deberá de indicar en el fichero *mvc-config.xml* el registro del message converter *udaMappingJacksonHttpMessageConverter*.

```
...
...
<bean id="requestMappingHandlerAdapter"
       class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

       <property name="messageConverters">
            <list>
...
...
               <ref bean="udaMappingJacksonHttpMessageConverter"/>
...
...
            </list>
       </property>

</bean>

...
...
```