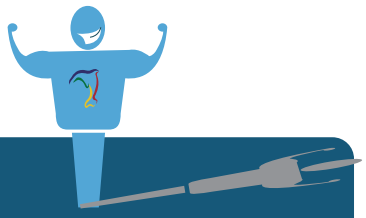
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

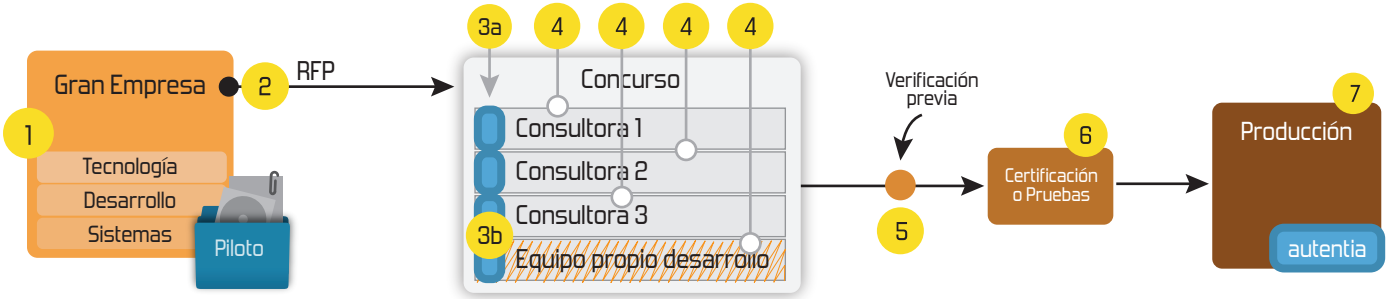
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Francisco J. Arroyo

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2012-09-20 Tutorial visitado 1 veces Descargar en PDF

Jugando con JSON en Java y la librería GSON. Parte 2

0. Índice de contenidos.

1. Introducción

Como hemos visto en el tutorial de Miguel Arlandy en Jugando con JSON en Java y la librería Gson podemos hacer montón de cosas con esta libreria, pero aún podemos hacer más!

Hay casos en los que necesitamos personalizar un poco más la serialización/deserialización ya que la traducción no es directa. Por poner un ejemplo (que usaré más adelante), quizás el receptor necesite recibir un 1 ó un 0 en vez de un true o un false.

También puede ser interesante, evitar serializar alguna propiedad de una clase. Por ejemplo, alguna propiedad en la que guardemos algún estado del objeto que sea temporal.

En este tutorial vamos a ver un ejemplo de estos dos casos, que a mí particularmente, me ha sido útil en los proyectos en los que he participado.

2. Entorno

- Hardware
  - Mackbook Pro
    - Intel Core i7 2Ghz
    - 8GB RAM
    - 500GB HD
    - Sistema Operativo: Mac OS X (10.8.2)
- Software
  - IntelliJ 11 Ultimate

3. Personalizar la serialización/deserialización.

Tanto para este ejemplo, como para el siguiente, vamos a utilizar el mismo modelo que ha usado Miguel en su tutorial. En este caso, en la clase SolicitudVacaciones hay una propiedad nueva "aceptadas" que usaremos para el primer ejemplo y la clase Empleado tiene otra nueva propiedad que es "campoAExcluir" que usaremos para el segundo ejemplo.

view plain print ?

```
01. public class Empleado {
02.
03.     private final int id;
04.
05.     private final String nombre;
06.
07.     private final String empresa;
08.
09.     private final List<SolicitudVacaciones> vacaciones;
10.
11.     @JsonExclude
12.     private String campoAExcluir = "";
13.
```

```
    public Empleado(int id, String nombre, String empresa, List<SolicitudVacaciones> vacaciones) {
        this.id = id;
        this.nombre = nombre;
        this.empresa = empresa;
        this.vacaciones = vacaciones;
    }

    //getters
}
```

view plain print ?

```
public class SolicitudVacaciones {

    private final Date inicio;

    private final Date fin;

    @SerializedName("d")
    private final int totalDias;

    //Se ha añadido el campo "aceptadas" para el ejemplo
    private Boolean aceptadas;

    public SolicitudVacaciones(Date inicio, Date fin, int totalDias) {
        this.inicio = inicio;
        this.fin = fin;
        this.totalDias = totalDias;
        this.aceptadas = Boolean.FALSE;
    }

    //gettes y setter
}
```

Si serializamos una instancia de la clase SolicitudVacaciones, obtendremos algo como esto:

view plain print ?

```
{
    "inicio":"Sep 18, 2012 10:34:13 AM",
    "fin":"Sep 18, 2012 10:34:13 AM",
    "d":0
    ,"aceptadas":false
}
```

Pero ¿Que ocurre si el receptor necesita recibir un 1 o un 0 en vez de true o false?. Tenemos varias opciones, como puede ser crear otro campo, hacer un wrapper de este objeto mapeando las propiedades a los campos que necesite el receptor, etc. Para mi, la opción más adecuada, es la personalización de la serialización, y esto es algo que **GSon** permite hacer de forma muy sencilla.

Si lo que queremos es personalizar la serialización, es decir, pasar de nuestro Objeto a JSON, debemos crearnos una clase que defina la serialización. El único requisito de la clase es que implemente JsonSerializer.

A continuación vemos la implementación de nuestro **BooleanTypeAdapter**

view plain print ?

```
public class BooleanTypeAdapter implements JsonSerializer<Boolean> {

    @Override
    public JsonElement serialize(Boolean aBoolean, Type type, JsonSerializationContext jsonSerializat
        if(aBoolean == null) {
            return new JsonPrimitive("");
        }
        return new JsonPrimitive(aBoolean.booleanValue() == true ? 1 : 0);
    }
}
```

Como veis, la cosa no puede ser más sencilla. Como la clase implementa **JsonSerializer**, tenemos que implementar el método **serialize**, que más tarde usará gson para serializar usando nuestra implementación. Si os fijáis en el cuerpo del método, lo único que hace es comprobar si el valor recibido es un nulo, que en ese caso devuelve una cadena vacía. En otro caso, mira el contenido de la variable y devuelve un 1 o un 0 según corresponda.

El segundo paso es indicar al builder, que nos construya una instancia de Gson que utilice nuestro serializador. Esto lo hacemos usando el método **registerTypeAdapter** indicando que serializador (TypeAdapter) queremos usar y para que tipo.

view plain print ?

```
final Gson gson = new GsonBuilder().setDateFormat("dd/MM/yyyy").registerTypeAdapter(Boolean.class, n
```

Si ejecutamos el siguiente test, veremos que nos da un bonito verde :). Fijaros que la última propiedad (aceptadas) devuelve un 0 y no un false.

view plain print ?

```
@Test
    public void shouldSerializeCustomBoolean() {
        final SolicitudVacaciones solicitudVacaciones = new SolicitudVacaciones(date, date, 0);
        final Gson gson = new GsonBuilder().setDateFormat("dd/MM/yyyy").registerTypeAdapter(Boolean.
        assertEquals("
{\"inicio\":\"01/01/2012\",\"fin\":\"01/01/2012\",\"d\":0,\"aceptadas\":0}", gson.toJson(solicitudVa
    }
```

Si lo que necesitamos es el proceso inverso, es decir, si al deserializar necesitamos un true o un false y lo que recibimos es un 0 lo podemos hacer de la misma manera. La única diferencia es que en vez de implementar Serializer debemos implementar **Deserializer**. Os pongo como quedaría implementada la clase anterior añadiendo el método para deserializar.

view plain print ?

```
public class BooleanTypeAdapter implements JsonSerializer<Boolean>, JsonDeserializer<Boolean> {

    //implementación JsonSerializer
```


» Android Beam

» Como hacer nuestros test más legibles con Hamcrest

» Ejemplo de Viewpager para android

» Uso de las anotaciones @Embeddable, @Embedded, @AttributeOverrides, @AssociationOverrides

» Como intentar averiguar el juego de caracteres de un archivo

```
    public JsonElement serialize(Boolean aBoolean, Type type, JsonSerializationContext jsonSerializat

     @Override
    public Boolean deserialize(JsonElement jsonElement, Type type, JsonDeserializationContext jsonDes
        int primitive = jsonElement.getAsJsonPrimitive().getAsInt();
        return primitive == 1 ? Boolean.TRUE : Boolean.FALSE;
    }

}
```

y el test que valida nuestra implementación

```
view plain  print  ?
@Test
    public void shouldDeSerializeCustomBoolean() {
        final Gson gson = new GsonBuilder().setDateFormat("dd/MM/yyyy").registerTypeAdapter(Boolean.
        final String solicitudVacacionesAceptadasFalse = "
{\"inicio\":\"01/01/2012\",\"fin\":\"01/01/2012\",\"d\":0,\"aceptadas\":0}";
        final String solicitudVacacionesAceptadasTrue = "
{\"inicio\":\"01/01/2012\",\"fin\":\"01/01/2012\",\"d\":0,\"aceptadas\":1}";
        assertEquals(Boolean.FALSE, gson.fromJson(solicitudVacacionesAceptadasFalse, SolicitudVacaci
        assertEquals(Boolean.TRUE, gson.fromJson(solicitudVacacionesAceptadasTrue, SolicitudVacacion
    }
```

4. Personalizar la estrategia de exclusión

Como comentaba en la introducción, hay ocasiones en los que quizás no interese serializar todas las propiedades de una clase. Por ejemplo una propiedad que guardemos un estado temporal/interno de la instancia, un valor que sea importante en el negocio de la aplicación pero no tenga sentido que se serialice.

Para conseguir evitar que serializemos propiedades, **Gson** permite hacerlo de manera muy sencilla. Necesitaremos crear una anotación, una clase e indicarle al constructor de Gson que utilice nuestra estrategia de exclusión.

La anotación la necesitamos para marcar las propiedades que queremos que Gson ignore.

```
view plain  print  ?
package com.pso.samsung.rest.provider.serializer;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface JsonExclude {
    //
}
```

Como vemos, la anotación no tiene nada de especial. Es una anotación que se usa en tiempo de ejecución y que está limitada a las propiedades de una clase. Esto último es importante porque Gson, al serializar un objeto, sólo mira las propiedades de la clase (no usa los getters/setters).

Una vez que tenemos nuestra anotación para marcar las propiedades, vamos a implementar nuestra estrategia de exclusión. Para ello, creamos una clase que implemente **ExclusionStrategy** e implementamos los dos métodos que nos proporciona la interfaz.

```
view plain  print  ?
public class JsonExclusionEstrategy implements ExclusionStrategy{

    @Override
    public boolean shouldSkipClass(Class <?> clazz) {
        return false;
    }

    @Override
    public boolean shouldSkipField(FieldAttributes fieldAttributes) {
        return fieldAttributes.getAnnotation(JsonExclude.class) != null;
    }

}
```

Con el primer método, decidimos si queremos excluir la clase entera en el momento de la serialización. En nuestro caso como solo queremos ignorar propiedades, devolvemos siempre false. El segundo método indica si debemos excluir una propiedad. Lo que está haciendo es mirar las propiedades de la propiedad que está serializando, y si la propiedad contiene la anotación que hemos creado en el paso anterior, entonces decimos que ignore esa propiedad

Por último, al igual que en los casos anteriores, le indicamos al constructor de Gson que utilice nuestra estrategia de exclusión

```
view plain  print  ?
final Gson gson = new GsonBuilder().setExclusionStrategies(new JsonExclusionEstrategy()).create();
```

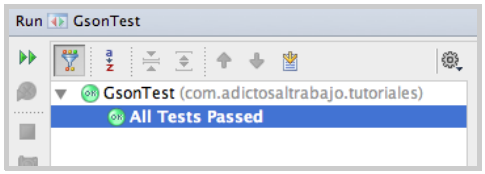
Hemos anotado la propiedad campoAExcluir con la anotación que hemos creado.

```
view plain  print  ?
public class Empleado {

    private final int id;

    private final String nombre;

    private final String empresa;

    private final List<SolicitudVacaciones> vacaciones;

    @JsonExclude
```

```
12.     private String campoAExcluir = "este valor nunca se va a serializar";
13.
14.     public Empleado(int id, String nombre, String empresa, List<SolicitudVacaciones> vacaciones) {
15.         this.id = id;
16.         this.nombre = nombre;
17.         this.empresa = empresa;
18.         this.vacaciones = vacaciones;
19.     }
20.
21.     //getters
22. }
```

El siguiente test comprueba dos cosas. Primero que el objeto que vamos a serializar tiene la propiedad "campoAExcluir" con un valor inicial. La segunda es que al serializar el objeto, no aparece el contenido de la propiedad que hemos excluido.

```
view plain  print  ?
01. @Test
02.     public void shouldExcludeFields() {
03.         final Gson gson = new GsonBuilder().setExclusionStrategies(new JsonExclusionEstrategy()).crea
04.         final Empleado empleado = new Empleado(0, "Mosca de autentia", "Autentia", null);
05.         assertEquals("este valor nunca se va a serializar", empleado.getCampoAExcluir());
06.         assertEquals("
{\"id\":0,\"nombre\":\"Mosca de autentia\",\"empresa\":\"Autentia\"}", gson.toJson(empleado));
07.     }
```



5. Conclusiones

Poco más que añadir a las conclusiones de Miguel

Es una librería muy sencilla de usar y muy util debido a la gran cantidad de APIs REST que hay y están surgiendo. Nos simplifica de una manera muy cómoda el tratamiento de JSON desde nuestras aplicaciones, tanto la serialización como la deserialización. Además no es especialmente pesada, la versión que hemos utilizado en el tutorial ocupa alrededor de 130kbs

Para cualquier comentario, duda o sugerencia, tenéis el formulario que aparece a continuación.

Un saludo.