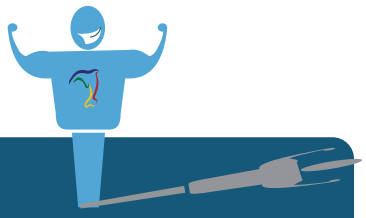
autentia

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...

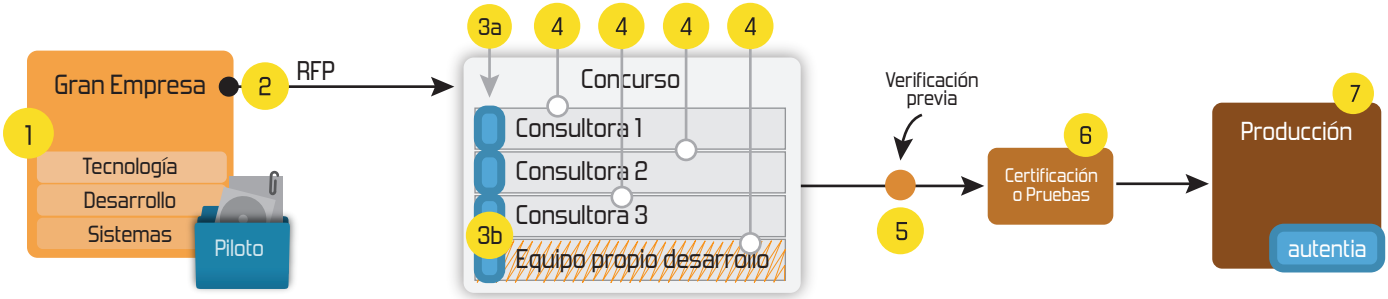
1. Desarrollo de componentes y proyectos a medida



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

Miguel Arlandy Rodríguez

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: 2012-09-17

Jugando con JSON en Java y la librería Gson.

0. Índice de contenidos.

1. Introducción

JSON (JavaScript Object Notation) es un formato de intercambio de información que está basado en estructuras de pares clave-valor. Es un formato mucho más ligero que XML y más indicado que éste en determinados escenarios (ojo!!! que no estoy diciendo que sea mejor, simplemente son distintos).

Puede darse la situación de que en nuestra aplicación Java, necesitemos atender peticiones representadas en JSON, transformarlas a Java, tratar los datos y devolver una respuesta en JSON. Los servicios REST o los Websockets son un buen ejemplo de esto.

Para resolver este problema podemos implementar "a mano" la lógica de negocio para serializar y deserializar nuestro JSON, lo que supodrá un esfuerzo considerable. O podríamos hacer uso de alguna librería diseñada para este propósito como puede ser Gson.

Y alguno dirá: "Pues yo uso Spring MVC o Jersey y lo hacen automáticamente". Cierto, pero porque utilizan por debajo liberías de este tipo (normalmente Jackson).

En este tutorial veremos cómo convertir objetos Java en objetos JSON y viceversa de manera muy sencilla gracias a la librería open-source Gson.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Mountain Lion 10.8
- Entorno de desarrollo: Intellij Idea 11.1 Ultimate.
- Gson 2.2.2
- Maven 3.0.3
- JUnit 4.8.2

3. Ejemplos.

Antes de nada necesitaremos añadir la siguiente dependencia a nuestra aplicación:

```
<dependency>
    <groupid>com.google.code.gson</groupid>
    <artifactid>gson</artifactid>
    <version>2.2.2</version>
</dependency>
```

El que no sea muy amigo de Maven puede descargarse la libería aquí.

El uso de esta librería **se basa en el uso de una instancia de la clase Gson**. Dicha instancia se puede crear de manera directa (new Gson()) para transformaciones sencillas o de forma más compleja con **GsonBuilder** para añadir distintos comportamientos. Lo veremos en los ejemplos.

Una instancia de la clase Gson **no mantiene ningún tipo de estado**, por lo que el mismo objeto puede reutilizarse para múltiples serializaciones/deserializaciones.

A continuación veremos una serie de ejemplos que nos ayudarán a entender cómo usar esta libería.

3.1. Deserializando JSON a un objeto Properties.

En este primer ejemplo (test) vamos a ver cómo deserializar nuestro objeto JSON en un objeto Properties (Java). Nuestro objeto JSON es el siguiente:

```
{
    "id" : 46,
    "nombre": "Miguel",
    "empresa": "Autentia"
}
```

Deserializar este objeto en un Properties (java.util.Properties) es muy sencillo con Gson. Basta con crear un objeto Gson e invocar a su **método fromJson**. Como parámetros le pasaremos el objeto JSON como String y la clase del objeto en que se deserializará. El siguiente test muestra cómo sería:

```
@Test
public void debeDevolverJSONEnUnProperties() {
    final String json = "{\"id\":46,\"nombre\":\"Miguel\",\"empresa\":\"Autentia\"}";
    final Gson gson = new Gson();
    final Properties properties = gson.fromJson(json, Properties.class);
    assertEquals("46", properties.getProperty("id"));
    assertEquals("Miguel", properties.getProperty("nombre"));
    assertEquals("Autentia", properties.getProperty("empresa"));
    assertNull(properties.getProperty("propiedadInexistente"));
}
```

Más fácil imposible... :-)

3.2. Deserializando JSON a un objeto propio.

En este ejemplo vamos a hacer lo mismo que en el anterior pero deserializando el objeto JSON en un objeto propio. La clase sería la siguiente:

```
public class Empleado {

    private final int id;

    private final String nombre;

    private final String empresa;

    public Empleado(int id, String nombre, String empresa) {
        this(id, nombre, empresa, null);
    }

    // Aquí los métodos get

}
```

Vemos que **los atributos de nuestra clase: id, nombre y empresa, coinciden con las claves del objeto JSON**. En posteriores ejemplos veremos que esto no tiene por qué ser necesariamente así. La forma de realizar la transformación es exactamente igual que la del ejemplo anterior, pero ahora al método fromJson le pasamos la clase Empleado en vez de Properties:

```
@Test
public void debeDevolverJSONEnUnObjeto() {
    final String json = "{\"id\":46,\"nombre\":\"Miguel\",\"empresa\":\"Autentia\"}";
    final Gson gson = new Gson();
    final Empleado empleado = gson.fromJson(json, Empleado.class);
    assertEquals(46, empleado.getId());
    assertEquals("Miguel", empleado.getNombre());
    assertEquals("Autentia", empleado.getEmpresa());
}
```

3.3. Serializando nuestro objeto propio en JSON.

Autor

» WebSockets con Java y Tomcat 7

» Introducción a Apache ActiveMQ

» Transiciones y animaciones con CSS3

» Comparando diferencias entre ficheros con java-diff-utils

» JQuery Waypoints: realizando acciones al llegar a un punto de la página con el scroll.

Categorías del Tutorial

Java Estándar

Otras Técnicas

Últimas ofertas de empleo

2011-09-08 Comercial - Ventas - MADRID.

2011-09-03 Comercial - Ventas - VALENCIA.

2011-08-19 Comercial - Compras - ALICANTE.

2011-07-12 Otras Sin catalogar - MADRID.

2011-07-06 Otras Sin catalogar - LUGO.

Ahora vamos a ver el ejemplo contrario al anterior. Partiendo de nuestro objeto empleado, queremos obtener su representación JSON. Para ello, igual que en los dos ejemplos anteriores, creamos una instancia de Gson. Esta vez invocamos a su **método toJSON** al que le pasaremos el objeto que queremos serializar en JSON:

```
@Test
public void debeDevolverLaRepresentacionJSONDeUnObjeto() {
    final Empleado empleado = new Empleado(46, "Miguel", "Autentia");
    final Gson gson = new Gson();
    final String representacionJSON = gson.toJson(empleado);
    assertEquals("{\"id\":46,\"nombre\":\"Miguel\",\"empresa\":\"Autentia\"}", representacionJSON);
}
```

3.4. Serializando nuestro objeto propio en JSON "bonito".

Si nos fijamos bien en el ejemplo anterior, la representación JSON de nuestro "Empleado" viene bastante comprimida. Todo en una línea y sin espacios o tabulaciones. Es posible que en algunos casos queramos mostrar la representación JSON de una forma más clara, por ejemplo en un fichero de log. Si lo hiciésemos de la forma del ejemplo anterior y si el objeto fuese más complejo es muy probable que nos costase interpretar la información.

Gson nos permite crear representaciones JSON un poco más vistosas. Para ello debemos crear una instancia de Gson con **GsonBuilder**. Activamos el modo PrettyPrinting invocando al método **setPrettyPrinting** y creamos la instancia con el método **create**.

```
@Test
public void debeDevolverLaRepresentacionJSONDeUnObjetoDeFormaBonita() {
    final Empleado empleado = new Empleado(46, "Miguel", "Autentia");
    final Gson prettyGson = new GsonBuilder().setPrettyPrinting().create();
    final String representacionBonita = prettyGson.toJson(empleado);
    final String representacionEsperada = "{\n" +
        "  \"id\": 46,\n" +
        "  \"nombre\": \"Miguel\",\n" +
        "  \"empresa\": \"Autentia\"\n"+
        "}";
    assertEquals(representacionEsperada, representacionBonita);
}
```

El resultado contiene espacios en blanco y saltos de línea, lo que facilita notablemente su lectura.

3.5. Deserializando JSON en una lista de objetos propios.

Este ejemplo es parecido al segundo pero con la salvedad de que ahora no convertimos un objeto JSON en un objeto Java propio, sino en una lista de éstos. El array de objetos JSON es el siguiente:

```
[
    {
        "id" : 46,
        "nombre": "Miguel",
        "empresa": "Autentia"
    },
    {
        "id" : 76,
        "nombre": "CR7",
        "empresa": "Real Madrid C.F"
    }
]
```

Como vemos hay dos empleados en el array. Para que Gson comprenda que tiene que deserializar el objeto JSON en una lista de objetos propios hacemos lo mismo que en los ejemplos anteriores. La única diferencia es que ahora el segundo parámetro del método fromJSON no es la clase a la que queremos deserializar el objeto JSON sino un objeto Type (java.lang.reflect.Type) que habremos creado mediante **TypeToken**. Al crear una instancia de TypeToken, lo tipamos con la lista de "Empleado", invocamos a su método **getType** y ya tenemos nuestra instancia de Type.

```
@Test
public void debeDevolerLaRepresentacionJSONEnUnaListaDeObjetos() {
    final String empleado1JSON = "{\"id\":46,\"nombre\":\"Miguel\",\"empresa\":\"Autentia\"}";
    final String empleado2JSON = "{\"id\":76,\"nombre\":\"CR7\",\"empresa\":\"Real Madrid C.F\"}";
    final String empleadosJSON = "[" + empleado1JSON + "," + empleado2JSON + "]";
    final Gson gson = new Gson();
    final Type tipoListaEmpleados = new TypeToken<List<Empleado>>(){}.getType();
    final List<Empleado> empleados = gson.fromJson(empleadosJSON, tipoListaEmpleados);
    assertNotNull(empleados);
    assertEquals(2, empleados.size());
    final Empleado empleado1 = empleados.get(0);
    final Empleado empleado2 = empleados.get(1);
    assertEquals(46, empleado1.getId());
    assertEquals("Miguel", empleado1.getNombre());
    assertEquals("Autentia", empleado1.getEmpresa());
    assertEquals(76, empleado2.getId());
    assertEquals("CR7", empleado2.getNombre());
    assertEquals("Real Madrid C.F", empleado2.getEmpresa());
}
```

3.6. Deserializando JSON en un objeto genérico.

Como hemos visto en el ejemplo anterior Gson es capaz de deserializar un objeto JSON en una lista tipada. Por supuesto esto es extensible a cualquier otra clase genérica. Veamos un ejemplo. Supongamos que tenemos la clase **Envoltorio**:

```
public class Envoltorio<T> {

    private final T objeto;

    public Envoltorio(T objeto) {
        this.objeto = objeto;
    }

```

```
    //getObjeto
}
```

Para deserializar un objeto JSON en una de estas clases con tipo genérico, lo hacemos de la misma forma que en el ejemplo anterior. Usamos un "TypeToken":

```
@Test
public void debeDevolerLaRepresentacionJSONDeUnTipoGenerico() {
    final String envoltorioGenericoJSON = "{\"objeto\":{\"id\":46,\"nombre\":\"Miguel\",\"empresa\
    final Type tipoEnvoltorioEmpleado = new TypeToken<Envoltorio<Empleado>>(){}.getType();
    final Gson gson = new Gson();
    final Envoltorio<Empleado> envoltorioEmpleado = gson.fromJson(envoltorioGenericoJSON, tipoEnvc
    assertEquals(46, envoltorioEmpleado.getObjeto().getId());
    assertEquals("Miguel", envoltorioEmpleado.getObjeto().getNombre());
    assertEquals("Autentia", envoltorioEmpleado.getObjeto().getEmpresa());
}
```

3.7. Deserializando fechas de JSON a un objeto propio.

En este ejemplo vamos a ver cómo transformar fechas de nuestro objeto JSON en fechas de nuestro objeto Java (java.util.Date). Nuestro objeto JSON representa un intervalo de fechas (lo usaremos para simular una solicitud de vacaciones). Es el siguiente:

```
{
    "inicio" : "06/08/2012",
    "fin" : "10/08/2012"
}
```

La clase Java que representará el intervalo de fechas será la siguiente:

```
public class SolicitudVacaciones {

    private final Date inicio;

    private final Date fin;

    public SolicitudVacaciones(Date inicio, Date fin) {
        this.inicio = inicio;
        this.fin = fin;
    }

    // métodos get
}
```

Para transformar las fechas lo haremos de la misma forma que en los ejemplos anteriores pero, al instanciar nuestro objeto Gson, lo haremos como lo hicimos en el ejemplo 4 con GsonBuilder. Inicializamos el formato de fecha con el método **setDateFormat** (en nuestro caso dd/MM/yyyy) y creamos la instancia con el método create. Lo vemos en el siguiente test:

```
@Test
public void debeTransformarLasFechas() throws ParseException {
    final String FORMATO_FECHA = "dd/MM/yyyy";
    final DateFormat DF = new SimpleDateFormat(FORMATO_FECHA);
    final String vacacionesJSON = "{\"inicio\":\"06/08/2012\",\"fin\":\"10/08/2012\"}";
    final Gson gson = new GsonBuilder().setDateFormat(FORMATO_FECHA).create();
    final SolicitudVacaciones vacaciones = gson.fromJson(vacacionesJSON, SolicitudVacaciones.class
    assertEquals(DF.parse("06/08/2012"), vacaciones.getInicio());
    assertEquals(DF.parse("10/08/2012"), vacaciones.getFin());
}
```

3.8. Mapeando propiedades del objeto JSON al objeto Java.

Como vimos en el ejemplo 2, si realizamos la deserialización del objeto JSON al objeto Java directamente con "fromJson", ésta se realizará propiedad a propiedad entre el objeto JSON y el Java de manera automática si las propiedades y atributos tienen el mismo nombre. Sin embargo, podemos hacer que esto no sea así. Vamos a añadir una propiedad nueva a nuestro objeto JSON del ejemplo anterior.

```
{
    "inicio" : "06/08/2012",
    "fin" : "10/08/2012",
    "d" : 5
}
```

La nueva propiedad "d" simboliza el número de días del intervalo de tiempo. Imaginemos que queremos mapearlo en un nuevo atributo "totalDias" en nuestra clase SolicitudVacaciones por temas de claridad de código. Para ello contamos con la anotación **SerializedName**:

```
public class SolicitudVacaciones {

    private final Date inicio;

    private final Date fin;

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

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

    // métodos get
}
```

Y lo probamos de la misma forma que hicimos con el ejemplo 2:

```
@Test
public void debeAsignarElValorAUnAtributoDeDistintoNombre() throws ParseException {
    final String vacacionesJSON = "{\"d\":5}";
    final gson = new Gson();
```

```
    final SolicitudVacaciones vacaciones = gson.fromJson(vacacionesJSON, SolicitudVacaciones.class)
    assertEquals(5, vacaciones.getTotalDias());
}
```

3.9. Deserializando un objeto JSON con una lista de objetos en un objeto Java.

Vamos a hacer una especie de "mix" entre el ejemplo 4 y el 5. A nuestra clase Empleado le vamos a añadir una colección de solicitud de vacaciones.

Añadimos las solicitudes de vacaciones a nuestro JSON:

```
{
    "id" : 46,
    "nombre" : "Miguel",
    "empresa" : "Autentia",
    "vacaciones":[
                    {
                        "inicio" : "06/08/2012",
                        "fin" : "10/08/2012",
                        "d" : 5
                    },
                    {
                        "inicio" : "23/08/2012",
                        "fin" : "29/08/2012",
                        "d" : 7
                    }
                ]
}
```

Y a nuestra clase Empleado:

```
public class Empleado {

    private final int id;

    private final String nombre;

    private final String empresa;

    private final List<SolicitudVacaciones> vacaciones;

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

    // métodos get

}
```

¿Y qué hay que hacer para mapear la lista de "SolicitudVacaciones"? Pues absolutamente nada fuera de lo que hemos visto. Gson lo hará de manera automática por nosostros:

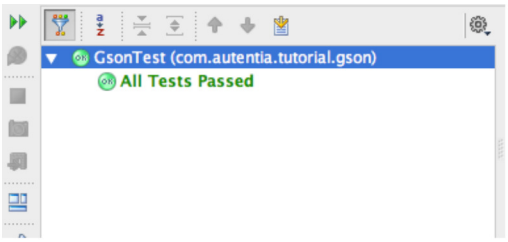
```
@Test
public void debeTransformarLaListaDeVacacionesDelEmpleado() {
    final String empleadoJSON = "{\"id\":46,\"nombre\":\"Miguel\",\"empresa\":\"Autentia\",\"vacac
    final Gson gson = new GsonBuilder().setDateFormat("dd/MM/yyyy").create();
    final Empleado empleado = gson.fromJson(empleadoJSON, Empleado.class);
    assertNotNull(empleado.getVacaciones());
    assertEquals(2, empleado.getVacaciones().size());
    assertEquals(5, empleado.getVacaciones().get(0).getTotalDias());
    assertEquals(7, empleado.getVacaciones().get(1).getTotalDias());
}
```

3.10. Leyendo JSON desde un fichero.

Nuestro objeto Gson no solo acepta un String como representación JSON del objeto a deserializar. También puede leerlo desde un fichero y realizar la mista tarea. Imaginemos que tenemos un fichero **json.txt** con el objeto JSON del ejemplo anterior. Lo único que tenemos que hacer es pasar a nuestro objeto Gson un Reader (java.io.Reader) con la información de ese fichero y Gson hará el resto:

```
@Test
public void debeTransformarLaListaDeVacacionesDelEmpleadoDesdeFichero() throws IOException {
    final Gson gson = new GsonBuilder().setDateFormat("dd/MM/yyyy").create();
    final InputStream is = GsonTest.class.getClassLoader().getResourceAsStream("json.txt");
    final BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(is));
    final Empleado empleado = gson.fromJson(bufferedReader, Empleado.class);
    assertNotNull(empleado.getVacaciones());
    assertEquals(2, empleado.getVacaciones().size());
    assertEquals(5, empleado.getVacaciones().get(0).getTotalDias());
    assertEquals(7, empleado.getVacaciones().get(1).getTotalDias());
    bufferedReader.close();
}
```

Pues yo creo que con todos estos ejemplos ya tenemos para hacernos una buena idea de lo fácil que es trabajar con JSON y Java gracias a Gson. Lanzamos los tests y todo perfecto... :-)



4. Referencias.

- Gson user guide

5. Conclusiones.

En este tutorial os hemos presentado la libería Gson, una libería excelente para trabajar con objetos JSON y Java. En Autentia ya nos hemos apoyado en esta libería en otros tutoriales que consumían y generaban JSON en distintos escenarios como son los Websockets y los Servicios REST.

En definitiva una libería buena, bonita y barata (tanto que es gratis) para trabjar con JSON y Java :-).

Para el que no esté todavía muy convencido con Gson y quiera buscar otras alternativas, le dejo el enlace de este tutorial donde nuestro compañero Alejandro nos presenta Jackson (una libería de propósito similar a Gson).

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy