


Open · Opened 3 days ago by  **Rubén Montero**

Una compañía lucrativa

Resumen

- Crearemos dos POJOs que tendrán un método destinado a *producir una representación* en JSON
- Los usaremos en un nuevo *endpoint*

Descripción

Para producir una respuesta JSON desde nuestro servidor, hemos escrito código parecido a este en nuestras últimas tareas:

```
@Get
public StringRepresentation getEndpointResponse() {
    JSONObject json = new JSONObject();
    // Aquí moldeo el objeto json a mi gusto
    // Añado claves-valor, etc.
    json.put("clave1", "valor 1 tipo String");
    json.put("clave2", 55) // numérico;
    String jsonString = json.toString();
    // Convertimos nuestro String a StringRepresentation
    StringRepresentation representation = new StringRepresentation(jsonString);
    representation.setMediaType(MediaType.APPLICATION_JSON);
    return representation;
}
```

Vamos a llevar a cabo una **mejora**.

Usaremos un *Plain Old Java Object* (**POJO**) para *encapsular* en una clase separada la lógica destinada a *producir un **String** resultado*.

En la práctica, la idea es llegar a algo como esto:

```
@Get
public StringRepresentation getEndpointResponse() {
    POJO pojo = new POJO("valor 1 tipo String", 55);
    String jsonString = pojo.toJSONObject().toString();
    // Convertimos nuestro String a StringRepresentation
    StringRepresentation representation = new StringRepresentation(jsonString);
    representation.setMediaType(MediaType.APPLICATION_JSON);
    return representation;
}
```

¿Por qué es una mejora?

Generar una respuesta de manera estática con `JSONObject` es poco práctico en la realidad.

Usar un **POJO** es el primer paso para conseguir que nuestra respuesta sea generada de manera *dinámica*. Profundizaremos en esto más adelante.

El objetivo de la tarea

Vamos a conseguir que en el *endpoint* `http://localhost:8104/company` se sirva un JSON como el siguiente:

```
{
  "name": "Amazon",
  "owner": "Jeff Bezos",
  "grossProfit": {
    "year": 2021,
    "amount": 54577000000,
    "currencyCodeIso4217": "USD"
  }
}
```

De momento los valores `Amazon`, `Jeff Bezos` serán *hardcodeados*.

Pero en un futuro próximo entenderemos la **utilidad** de emplear POJOs para encapsular los datos que se sirven.

¿Cuántos POJOs?

Usaremos **dos**.

Uno estará **dentro del otro**. El primero acumulará los valores del JSON raíz, y el segundo está destinado a englobar los valores del *objeto* dentro de `"grossProfit"`.

¿Eh?

Creemos nuestro **primer POJO**.

Añade una nueva clase `GrossProfit.java` que tenga **3** atributos privados:

```
public class GrossProfit {  
    private int year;  
    private long amount; // Can be a number bigger than 2,147,483,647 so we must use Long instead of int  
    private String currencyCode;  
  
}
```

Para esta tarea, basta con que tenga un método constructor:

```
public class GrossProfit {  
    private int year;  
    private long amount;  
    private String currencyCode;  
  
    public GrossProfit(int year, long amount, String currencyCode) {  
        this.year = year;  
        this.amount = amount;  
        this.currencyCode = currencyCode;  
    }  
}
```

...y tendrá también un método que **produzca un `JSONObject`**:

toJSONObject

La idea es:

1. Crear una nueva *instancia* de `JSONObject`
2. Meter en ella los datos relevantes, con las claves JSON apropiadas
3. Devolverla

```
public JSONObject toJSONObject() {  
    JSONObject jsonResult = new JSONObject();  
    jsonResult.put("year", this.year);  
    jsonResult.put("amount", this.amount);  
    jsonResult.put("currencyCodeIso4217", this.currencyCode);  
    return jsonResult;  
}
```

Encajaremos esta pieza con el resto más adelante.

Segundo POJO

Representará el JSON final.

O sea, tendrá estos atributos:

```
private String name;  
private String owner;  
private ??? grossProfit;
```

¿De **qué tipo** debe ser el atributo `grossProfit`?

Pues, efectivamente, para eso creamos **2** clases distintas.

Una **contiene a la otra**:

```
private String name;
private String owner;
private GrossProfit grossProfit;
```

Manos a la obra

Añade una nueva clase `Company` con los atributos mencionados y un constructor:

```
public class Company {
    private String name;
    private String owner;
    private GrossProfit grossProfit;

    public Company(String name, String owner, GrossProfit grossProfit) {
        this.name = name;
        this.owner = owner;
        this.grossProfit = grossProfit;
    }
}
```

toJSONObject aquí también

Nuestro `Company` debe poder convertirse fácilmente a un `JSONObject`.

Añadamos un nuevo método:

```
public JSONObject toJSONObject() {
    JSONObject jsonResult = new JSONObject();
    jsonResult.put("name", this.name);
    jsonResult.put("owner", this.owner);
    jsonResult.put("grossProfit", this.grossProfit); // (?)
    return jsonResult;
}
```

Un nuevo endpoint

Añadamos una nueva clase `JSONGetCompany`, análoga a la de la tarea anterior:

```
public class JSONGetCompany extends ServerResource {

    @Get
    public StringRepresentation getEndpointResponse() {
        // ...
    }
}
```

Usando Company y GrossProfit

En esta ocasión, queremos sacar partido a las clases que acabamos de crear.

Podemos instanciarlas con valores apropiados así:

```
public class JSONGetCompany extends ServerResource {

    @Get
    public StringRepresentation getEndpointResponse() {
        GrossProfit profit = new GrossProfit(
            2021,
            54577000000L, // En Java, Los Long terminan en "L"
            "USD"
        );
        Company company = new Company(
            "Amazon",
            "Jeff Bezos",
            profit // ¡Aquí pasamos una variable tipo GrossProfit!
        );
    }
}
```

```

    );
    // ...

}
}

```

En el **futuro** veremos la importancia de esto, pues, aunque ahora estos valores están *hardcodeados*, podrían provenir de otra fuente de datos 😊

Devolviendo un **StringRepresentation** como ya sabemos

Las líneas que faltan para completar el método están al inicio de la tarea y son siempre iguales:

```

public class JSONGetCompany extends ServerResource {

    @Get
    public StringRepresentation getEndpointResponse() {
        GrossProfit profit = new GrossProfit(
            2021,
            54577000000L, // En Java, Los Long terminan en "L"
            "USD"
        );
        Company company = new Company(
            "Amazon",
            "Jeff Bezos",
            profit // ¡Aquí pasamos una variable tipo GrossProfit!
        );
        String jsonString = company.toJSONString().toString();
        StringRepresentation representation = new StringRepresentation(jsonString);
        representation.setMediaType(MediaType.APPLICATION_JSON);
        return representation;
    }
}

```

El endpoint... ¡y listo!

Basta con añadir esta línea a `SimpleREST.java` :

```
host.attach("/company", JSONGetCompany.class);
```

Visitamos `http://localhost:8104/company` desde el navegador y...

Ups...

Parece que el JSON entregado no está *del todo* bien:

```

{
  "owner": "Jeff Bezos",
  "name": "Amazon",
  "grossProfit": "GrossProfit@72d74e70"
}

```

¿Por qué **"grossProfit"** se ve mal?

El problema se encuentra en el método `toJSONObject()` dentro de `Company.java` .

Allí, pusimos una línea como esta:

```
jsonResult.put("grossProfit", this.grossProfit); // (?)
```

Pero `.put()` es peligroso y nos ha llevado a error.

¿Por qué?

`.put()` admite *varios tipos* de variable como **segundo** parámetro:

```

jsonEjemplo.put("clave", "Dato tipo String");
jsonEjemplo.put("clave", 5555); // int

```

```
jsonEjemplo.put("clave", 6549815698L); // Long
jsonEjemplo.put("clave", otroJsonObject);
```

Es decir, el método `.put` está **sobrecargado**.

Nosotros hemos invocado `.put` enviando un *objeto cualquiera* (`GrossProfit`), y Java, en su mejor esfuerzo, **lo ha convertido a `String` automáticamente**.

Al convertirlo, ha invocado [un `toString\(\)` por defecto](#) que tienen **todos** los objetos y que produce un resultado como el visto:

```
"GrossProfit@72d74e70"
```

¡Y está mal!

¿Qué hacemos?

Debemos invocar `.put()` pasando un `JSONObject` como **segundo** argumento, y **no** un `String`.

¿Cómo?

¡Fácil!

Prueba con:

```
- jsonResult.put("grossProfit", this.grossProfit);
+ jsonResult.put("grossProfit", this.grossProfit.toJSONObject());
```

Por último

Verifica que el *endpoint* `/company` se ve de la forma esperada y que el *test* funciona correctamente.

Haz `commit` y `push` para subir los cambios al repositorio.



[Rubén Montero @ruben.montero](#) changed milestone to [%Sprint 2](#) 3 days ago