(/)

Introducción al Jackson ObjectMapper

Última modificación: 17 de abril de 2019

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Datos (https://www.baeldung.com/category/data/)
Jackson (https://www.baeldung.com/category/json/jackson/)
JSON (https://www.baeldung.com/category/json/)

Jackson Basics (https://www.baeldung.com/tag/jackson-basics/)

1. Información general

Este artículo se centra en comprender la clase Jackson *ObjectMapper* y en cómo serializar objetos Java en JSON y deserializar cadenas JSON en objetos Java. Para comprender más sobre la biblioteca de Jackson en general, el Tutorial de Jackson (/jackson) es un buen lugar para comenzar.

Otras lecturas:

Herencia con Jackson (https://www.baeldung.com/jackson-inheritance)

Este tutorial demostrará cómo manejar la inclusión de metadatos de subtipo e ignorar las propiedades heredadas de las superclases con Jackson.

Leer más (https://www.baeldung.com/jackson-inheritance) →

Jackson JSON Views (https://www.baeldung.com/jackson-json-view-annotation)

Cómo usar la anotación @JsonView en Jackson para controlar perfectamente la serialización de sus objetos (sin y con Spring).

Leer más (https://www.baeldung.com/jackson-json-view-annotation) →

Jackson: serializador personalizado (https://www.baeldung.com/jackson-custom-serialization)

Controle su salida JSON con Jackson 2 utilizando un serializador personalizado.

Leer más (https://www.baeldung.com/jackson-custom-serialization) →

2. Dependencias

Primero agreguemos las siguientes dependencias al *pom.xml* :

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.8</version>
</dependency>
```

Esta dependencia también agregará transitivamente las siguientes bibliotecas al classpath:

1. jackson-annotations-2.9.8.jar
2. jackson-core-2.9.8.jar
3. jackson-databind-2.9.8.jar

Utilice siempre las últimas versiones en el repositorio central de Maven para el enlace de datos Jackson (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.fasterxml.jackson.core%22%20AND%20a%3A%22jackson-databind%22) .

3. Leer y escribir usando *ObjectMapper*

Comencemos con las operaciones básicas de lectura y escritura.

La API *readValue* simple de *ObjectMapper* es un buen punto de entrada. Podemos usarlo para analizar o deserializar contenido JSON en un objeto Java.

Además, en el lado de la escritura, **podemos usar la API *writeValue* para serializar cualquier objeto Java como salida JSON** .

Utilizaremos la siguiente clase *Car* con dos campos como objeto para serializar o deserializar a lo largo de este artículo:

```
public class Car {

    private String color;
    private String type;

    // standard getters setters
}
```

3.1. Objeto Java para JSON

Veamos un primer ejemplo de serialización de un objeto Java en JSON usando el método *writeValue* de la clase *ObjectMapper*:

```
ObjectMapper objectMapper = new ObjectMapper();
Car car = new Car("yellow", "renault");
objectMapper.writeValue(new File("target/car.json"), car);
```

El resultado de lo anterior en el archivo será:

```
{"color":"yellow","type":"renault"}
```

Los métodos *writeValueAsString* y *writeValueAsBytes* de la clase *ObjectMapper* generan un JSON a partir de un objeto Java y devuelve el JSON generado como una cadena o como una matriz de bytes:

```
String carAsString = objectMapper.writeValueAsString(car);
```

3.2. JSON a objeto Java

A continuación se muestra un ejemplo simple de convertir una cadena JSON en un objeto Java utilizando la clase *ObjectMapper*:

```
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
Car car = objectMapper.readValue(json, Car.class);
```

La función *readValue ()* también acepta otras formas de entrada como un archivo que contiene una cadena JSON:

```
Car car = objectMapper.readValue(new File("target/json_car.json"), Car.class);
```

o una URL:

```
Car car = objectMapper.readValue(new URL("target/json_car.json"), Car.class);
```

3.3. JSON a Jackson *JsonNode*

Alternativamente, un JSON puede analizarse en un objeto *JsonNode* y usarse para recuperar datos de un nodo específico:

```
String json = "{ \"color\" : \"Black\", \"type\" : \"FIAT\" }";
JsonNode jsonNode = objectMapper.readTree(json);
String color = jsonNode.get("color").asText();
// Output: color -> Black
```

3.4. Crear una lista Java a partir de una cadena de matriz JSON

Podemos analizar un JSON en forma de matriz en una lista de objetos Java usando una *referencia de tipo:*

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" : \"Red\", \"type\" : \"FIAT\" }]";
List<Car> listCar = objectMapper.readValue(jsonCarArray, new TypeReference<List<Car>>(){});
```

3.5. Crear un mapa Java a partir de una cadena JSON

Del mismo modo, podemos analizar un JSON en un *mapa de* Java :

```
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
Map<String, Object> map
  = objectMapper.readValue(json, new TypeReference<Map<String,Object>>(){});
```

4. Características avanzadas

Una de las mayores fortalezas de la biblioteca Jackson es el proceso de serialización y deserialización altamente personalizable.

En esta sección, veremos algunas características avanzadas en las que la respuesta JSON de entrada o salida puede ser diferente del objeto que genera o consume la respuesta.

4.1. Configuración de la característica de serialización o deserialización

Al convertir objetos JSON en clases Java, en caso de que la cadena JSON tenga algunos campos nuevos, el proceso predeterminado generará una excepción:

```
String jsonString
  = "{ \"color\" : \"Black\", \"type\" : \"Fiat\", \"year\" : \"1970\" }";
```

La cadena JSON en el ejemplo anterior en el proceso de análisis predeterminado del objeto Java para el *Class Car* dará como resultado la excepción *UnrecognizedPropertyException* .

A través del método de *configuración* podemos extender el proceso predeterminado para ignorar los nuevos campos:

```
objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
Car car = objectMapper.readValue(jsonString, Car.class);

JsonNode jsonNodeRoot = objectMapper.readTree(jsonString);
JsonNode jsonNodeYear = jsonNodeRoot.get("year");
String year = jsonNodeYear.asText();
```

Otra opción más se basa en *FAIL_ON_NULL_FOR_PRIMITIVES* que define si se permiten los valores *nulos* para los valores primitivos:

```
objectMapper.configure(DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES, false);
```

Del mismo modo, *FAIL_ON_NUMBERS_FOR_ENUM* controla si los valores de enumeración se pueden serializar / deserializar como números:


```
objectMapper.configure(DeserializationFeature.FAIL_ON_NUMBERS_FOR_ENUMS, false);
```

Puede encontrar la lista completa de características de serialización y deserialización en el sitio oficial (https://github.com/FasterXML/jackson-databind/wiki/Serialization-Features) .

4.2. Crear serializador o deserializador personalizado

Otra característica esencial de la clase *ObjectMapper* es la capacidad de registrar serializadores (/jackson-custom-serialization) y deserializadores personalizados (/jackson-deserialization) . El serializador y el deserializador personalizados son muy útiles en situaciones en las que la respuesta JSON de entrada o salida es diferente en estructura que la clase Java en la que se debe serializar o deserializar.

A continuación se muestra un ejemplo de serializador JSON personalizado:

```
public class CustomCarSerializer extends StdSerializer<Car> {

    public CustomCarSerializer() {
        this(null);
    }

    public CustomCarSerializer(Class<Car> t) {
        super(t);
    }

    @Override
    public void serialize(
      Car car, JsonGenerator jsonGenerator, SerializerProvider serializer) {
        jsonGenerator.writeStartObject();
        jsonGenerator.writeStringField("car_brand", car.getType());
        jsonGenerator.writeEndObject();
    }
}
```

Este serializador personalizado se puede invocar así:

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module =
  new SimpleModule("CustomCarSerializer", new Version(1, 0, 0, null, null, null));
module.addSerializer(Car.class, new CustomCarSerializer());
mapper.registerModule(module);
Car car = new Car("yellow", "renault");
String carJson = mapper.writeValueAsString(car);
```

Así es como se ve el *automóvil* (como salida JSON) en el lado del cliente:

```
var carJson = {"car_brand":"renault"}
```

Y aquí hay un ejemplo de **un deserializador JSON personalizado** :

```
public class CustomCarDeserializer extends StdDeserializer<Car> {

    public CustomCarDeserializer() {
        this(null);
    }

    public CustomCarDeserializer(Class<?> vc) {
        super(vc);
    }

    @Override
    public Car deserialize(JsonParser parser, DeserializationContext deserializer) {
        Car car = new Car();
        ObjectCodec codec = parser.getCodec();
        JsonNode node = codec.readTree(parser);

        // try catch block
        JsonNode colorNode = node.get("color");
        String color = colorNode.asText();
        car.setColor(color);
        return car;
    }
}
```

Este deserializador personalizado se puede invocar de la siguiente manera:

```
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
ObjectMapper mapper = new ObjectMapper();
SimpleModule module =
  new SimpleModule("CustomCarDeserializer", new Version(1, 0, 0, null, null, null));
module.addDeserializer(Car.class, new CustomCarDeserializer());
mapper.registerModule(module);
Car car = mapper.readValue(json, Car.class);
```

4.3. Formatos de fecha de manejo

La serialización predeterminada de *java.util.Date* produce un número, es decir, una marca de tiempo de época (número de milisegundos desde el 1 de enero de 1970, UTC). Pero esto no es muy legible para los humanos y requiere una mayor conversión para mostrarse en un formato legible para los humanos.

Vamos a ajustar la instancia de *Car* que utilizamos hasta ahora dentro de la clase *Request* con la propiedad *datePurchased* :

```
public class Request
{
    private Car car;
    private Date datePurchased;

    // standard getters setters
}
```

Para controlar el formato de cadena de una fecha y configurarlo, por ejemplo, *aaaa-MM-dd HH: mm az* , considere el siguiente fragmento:

```
ObjectMapper objectMapper = new ObjectMapper();
DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm a z");
objectMapper.setDateFormat(df);
String carAsString = objectMapper.writeValueAsString(request);
// output: {"car":{"color":"yellow","type":"renault"},"datePurchased":"2016-07-03 11:43 AM CEST"}
```

Para obtener más información sobre la serialización de fechas con Jackson, lea nuestro artículo más detallado (/jackson-serialize-dates) .

4.4. Manejo de colecciones

Otra característica pequeña pero útil disponible a través de la clase *DeserializationFeature* es la capacidad de generar el tipo de colección que queremos a partir de una respuesta JSON Array.

Por ejemplo, podemos generar el resultado como una matriz:

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" : \"Red\", \"type\" : \"FIAT\" }]";
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.configure(DeserializationFeature.USE_JAVA_ARRAY_FOR_JSON_ARRAY, true);
Car[] cars = objectMapper.readValue(jsonCarArray, Car[].class);
// print cars
```

O como una *lista* :

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" : \"Red\", \"type\" : \"FIAT\" }]";
ObjectMapper objectMapper = new ObjectMapper();
List<Car> listCar = objectMapper.readValue(jsonCarArray, new TypeReference<List<Car>>(){});
// print cars
```

Más información sobre el manejo de colecciones con Jackson está disponible aquí (/jackson-collection-array) .

5. Conclusión

Jackson es una biblioteca de serialización / deserialización JSON sólida y madura para Java. La API *ObjectMapper* proporciona una forma sencilla de analizar y generar objetos de respuesta JSON con mucha flexibilidad.

El artículo analiza las características principales que hacen que la biblioteca sea tan popular. El código fuente que acompaña al artículo se puede encontrar en GitHub. (https://github.com/eugenp/tutorials/tree/master/jackson-simple)

¡Los comentarios están cerrados en este artículo!

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)