# Sérialisation et représentation des données dans les services REST

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

## Table des matières

1 Formats d'échange de données ..... 1
2 JAXB (Jakarta XML Binding) ..... 1
3 JSON ..... 4
4 Autres formats ..... 7

## 1 Formats d'échange de données

Les services REST nécessitent l'échange de données entre client et serveur. Deux formats principaux dominent :

1. **XML (eXtensible Markup Language)**
   - Standard W3C
   - Format historique et très établi
   - Riche en fonctionnalités mais verbeux
2. **JSON (JavaScript Object Notation)**
   - Standard ECMA-404
   - Format léger et populaire
   - Bien qu'initialement non standardisé pour le web, est devenu le format de facto

Ces formats nécessitent des mécanismes de conversion (sérialisation/désérialisation) entre les objets Java et leur représentation textuelle. Pour cela, Java propose plusieurs APIs standards : - JAXB pour XML - Jackson ou JSON-B pour JSON

## 2 JAXB (Jakarta XML Binding)

Nous allons voir maintenant une introduction rapide au mapping XML↔Java. La définition des formats de données XML se fait par annotation des entités en utilisant le standard JAXB (Java Architecture for XML Binding) : `@XmlElement`, `@XmlType`, `@XmlAttribute`, `@XmlTransient`, `@XmlValue`, ...

Depuis Java 9, il est nécessaire d'ajouter les dépendances suivantes pour traiter des données XML avec JAXB :

La classe `Task` ci-dessous est un exemple simple. marquée comme étant représentée comme un élément XML (`@XmlRootElement`). On précise que les annotations sont faites sur les champs avec `@XmlAccessorType` (utile avec Lombok).

Par défaut, les propriétés sont représentées comme des éléments XML. Il est possible de préciser que l'on veut un attribut (`@XmlAttribute`) sur `id`, de contrôler leur nom (paramètre `name`) et de définir ceux qui ne doivent pas apparaitre (`@XMLTransient`). Il est aussi possible de contrôler l'ordre d'apparition des éléments (`propOrder` de `@XmlType`).

Attention, un constructeur sans paramètre (au maximum `protected`) est obligatoire (pour permettre la reconstruction). Sinon `@XmlType.factoryMethod()` et `@XmlType.factoryClass()` permettent d'utiliser une factory s'il s'agit d'une méthode statique sans paramètre.

Dans le cas d'une collection `@XmlElementWrapper` permet d'ajouter un élément parent au contenu et `@XmlElements` contrôle le type des éléments en fonction du type réel Java. Pour des primitifs `@XmlList` permet de générer des listes avec un espace comme séparateur.

`@XmlType` est similaire à `@XmlRootElement` si la classe ne doit apparitre que comme un sous-élément.

`@XmlValue` ne peut être utilisée que sur une seule propriété dont la valeur sera alors le contenu de l'élément (sans élément parent).

JAXB offre la classe JAXBContext pour transformer une classe Java en XML (Marshalling).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:task id="1" xmlns:ns2="http://bruno.univ-tln.fr/sample-jaxb/task">
    <status>1</status>
    <title>First task</title>
    <description>...</description>
    <tags>
        <tag>important</tag>
        <tag>outside</tag>
    </tags>
</ns2:task>
```

Le contexte JABX permet aussi simplement de réaliser l'opération inverse (UnMarshalling) à partir d'un document XML contenu dans une String, un fichier, d'un flux, ...

Dans le cas de JAX-RS, c'est le framework qui prend en charge la transformation des données retournées et reçues à condition d'ajouter la dépendance suivante en plus de celles de JAXB :

```
<dependency>
 <groupId>org.glassfish.jersey.media</groupId>
 <artifactId>jersey-media-jaxb</artifactId>
</dependency>
```

JAXB Permet aussi de générer automatique le Schema XML à partir des classes Java. Il suffit d'écrire une sous-classe de `SchemaOutputResolver` pour indiquer où les résultat doit être produit. Ci dessous deux exemples pour obtenir une String et des fichiers.

```
<?xml version="1.0" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="task">
    <xs:sequence>
      <xs:element name="status" type="state" minOccurs="0"/>
      <xs:element name="title" type="xs:string" minOccurs="0"/>
      <xs:element name="description" type="description" minOccurs="0"/>
      <xs:element name="tags" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
                      <xs:element  name="tag"  type="xs:string"  minOccurs="0"
maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:long" use="required"/>
  </xs:complexType>

  <xs:simpleType name="description">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

  <xs:simpleType name="state">
    <xs:restriction base="xs:int">
      <xs:enumeration value="1"/>
      <xs:enumeration value="0"/>
    </xs:restriction>
  </xs:simpleType>
```

```
</xs:schema>

<?xml version="1.0" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://bruno.univ-tln.fr/sample-jaxb/
task" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import schemaLocation="schema2.xsd"/>

  <xs:element name="task" type="task"/>

</xs:schema>
```

# 3 JSON

Le standard officiel pour JSON est maintenant JSON-B (Java API for JSON Binding). Cependant, des fonctionnalités importantes sont manquantes comme la gestion des types Polymorphes ou de certaines classes importantes en natif (comme les collections eclipses). Nous utiliserons donc une autre librairie : Jackson (cf. pom.xml).

Pour l'utiliser, il suffit d'ajouter les dépendances suivantes :

Le contrôle de la sérialisation/désérialisation se fait principalement par annotation des entités. Les méthodes de sérialisation/désérialisations sont alors générées automatiquement. Il est possible de créer manuellement ces méthodes pour un contrôle plus précis.

Les principales annotations sont données dans le tableau ci dessous et illustrées dans l'exemple suivant.

| annotation | Description |
|---|---|
| `@JsonProperty` | contrôle le nom d'une propriété. |
| `@JsonRootName` | défini les nom d'un élément « wrapper », doit être activée dans l'objectMapper. |
| `@JsonPropertyOrder` | défini l'ordre des propriétés. |
| `@JsonRawValue` | indique qu'une propriété contient du JSON et doit être utilisée sans conversion. |
| `@JsonValue` | indique la seule méthode qui retourne le contenu à serialiser. Nécessite un |

| annotation | Description |
|---|---|
| | constructeur avec un paramètre du même type pour la désérialisation. |
| `@JsonIgnore` (sur une propriété) | pour ignorer une ou plusieurs propriété. |
| `@JsonIgnoreProperties` (sur la classe) | pour ignorer une ou plusieurs propriété. |
| `@JsonIgnoreType` (sur la classe) | permet d'ignorer toutes les propriétés d'un type donné. |
| `@JsonUnwrapped` | inclut directement les propriétés d'un objet dans la classe qui le référence. |
| `@JsonInclude` | permet d'inclure ou d'ignorer les propriétés dont la valeur est nulle, vide ou celle par défaut. |

Pour les types polymorphes des annotations spécifiques pour les classes qui permettent d'indiquer comment le type est indiqué (`@JsonTypeInfo`), quels sont les sous-types (`@JsonSubTypes`) et le nom donné à chaque type (`@JsonTypeName`).

`@JsonView(XXX.class)` définit des vues différentes qui peuvent être choisir dans l'objectmapper.

La sérialisation/désérialisation est réalisée à l'aide d'une classe appelée ObjectMapper. Dans le cas de JAX-RS cette opération sera réalisée automatiquement par le framework.

```
[ {
  "Task" : {
    "id" : 1,
    "status" : "OPENED",
    "title" : "First task",
    "tags" : [ "important", "outside" ],
    "creationDate" : "2026-11-01@12:22:00"
  }
}, {
  "Task" : {
    "id" : 2,
    "status" : "OPENED",
    "title" : "Second task",
    "tags" : [ "optionnal" ],
    "creationDate" : "2026-11-01@12:22:00"
  }
} ]
```

La construction d'une object Java depuis JSON est très simple avec la methode `readValue` de `ObjectMapper`.

Jackson propose une gestion simple des références qui prend en compte les cycles. Dans notre exemple, si une tâche est associée à un utilisateur qui référence aussi toutes ses tâches, il y a une boucle infinie lors de la sérialisation. Une solution consiste à utiliser dans au moins l'un des deux l'indentifiant de l'autre.

L'annotation `@JsonIdentityInfo` permet de définir la solution pour identifier les instances d'une classe. Elle est alors utilisée automatiquement quand cela est nécessaire.

`@JsonIdentityReference(alwaysAsId = true)` permet de contrôler l'usage de l'identifiant (ici de le rendre systématique).

`@JsonBackReference` et `@JsonManagedReference` pour les références unidirectionnelles.

```
[ {
  "@type" : "User",
  "uuid" : "47e02447-5d5d-42c5-b6c1-ca0d8c4a1f1f",
  "name" : "John",
  "tasks" : [ "0db0d560-3229-4faf-b2fc-ab8ba3d9d1fa", "f5c84243-989f-46a8-b9a6-
ef9cc5cfe0c1" ]
}, {
  "@type" : "Task",
  "uuid" : "0db0d560-3229-4faf-b2fc-ab8ba3d9d1fa",
  "title" : "T1",
  "owner" : "47e02447-5d5d-42c5-b6c1-ca0d8c4a1f1f"
}, {
  "@type" : "Task",
  "uuid" : "f5c84243-989f-46a8-b9a6-ef9cc5cfe0c1",
  "title" : "T2",
  "owner" : "47e02447-5d5d-42c5-b6c1-ca0d8c4a1f1f"
} ]
```

La lecture de données JSON se fait de la même manière. Attention, pour le lien bidirectionnel il ne doit pas apparaitre deux fois (dans User et dans Task) sinon les données sont ajoutées en double.

```
[ {
  "@type" : "User",
  "uuid" : "487d6096-7608-11eb-9439-0242ac130002",
  "name" : "Mary",
            "tasks"      :      [      "5ba90634-7608-11eb-9439-0242ac130002",
"69b64aa2-7608-11eb-9439-0242ac130002" ]
```

```
}, {
  "@type" : "Task",
  "uuid" : "5ba90634-7608-11eb-9439-0242ac130002",
  "title" : "TM1",
  "owner" : "487d6096-7608-11eb-9439-0242ac130002"
}, {
  "@type" : "Task",
  "uuid" : "69b64aa2-7608-11eb-9439-0242ac130002",
  "title" : "TM2",
  "owner" : "487d6096-7608-11eb-9439-0242ac130002"
} ]
```

Pour aller plus loin, JSON n'est en fait pas un standard du Web et donc chaque format est « propriétaire ». JSON-LD qui s'appuie sur JSON pour représenter des données sémantiques sur le Web est une meilleure solution. Pour être, complètement compatible avec l'approche HATEOS, Un vocabulaire spécifique pour les API RESGT construit au dessus de JSON-LD appelé Hydra est en cours de définition. LE site schemas.org propose de standardiser des schémas courants.

## 4 Autres formats

Par curiosité, Jackson propose aussi d'autres formats comme YAML.

```
---
- !<User>
                                        &487d6096-7608-11eb-9439-0242ac130002
uuid: "487d6096-7608-11eb-9439-0242ac130002"
  name: "Mary"
  tasks:
  - "5ba90634-7608-11eb-9439-0242ac130002"
  - "69b64aa2-7608-11eb-9439-0242ac130002"
- !<Task>
                                        &5ba90634-7608-11eb-9439-0242ac130002
uuid: "5ba90634-7608-11eb-9439-0242ac130002"
  title: "TM1"
  owner: *487d6096-7608-11eb-9439-0242ac130002
- !<Task>
                                        &69b64aa2-7608-11eb-9439-0242ac130002
uuid: "69b64aa2-7608-11eb-9439-0242ac130002"
  title: "TM2"
  owner: *487d6096-7608-11eb-9439-0242ac130002
```