Jackson

Theory

17 minutes reading

JSON is the accepted standard these days for representing, storing, and transferring data so every developer should know how to work with it. Since the Java Standard library doesn't provide a way to convert Java objects to JSON and vice versa, we have to use third-party libraries like **Jackson**. Jackson is a powerful, lightweight, and flexible Java library that is used by default by many other popular frameworks such as Spring, ElasticSearch, OkHttp, Hadoop, and so on. Because it's so widely used, Jackson is a must-have tool for all Java software developers.

§1. Dependency

To work on our project with the Jackson library, you must add the `jackson-databind` dependency which contains two other necessary dependencies, namely `jackson-annotations` and `jackson-core`. Its latest version is currently 2.14.1.

You can always find the latest version on the Maven website.

▼ Gradle

```
implementation 'com.fasterxml.jackson.core:jackson-databind:2.14.1'
```

▼ Maven

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.14.1</version>
</dependency>
```

§2. Default serialization

Let's imagine that we are developing a Twitter-like app and one of the main entities is a **post**. We want to describe the main components of the post: id, creation date, content, number of likes, and a list of comments (this is a rather simplistic model, but it's enough for us now).

```
public class Post {
    private int id;
    private Date createdDate;
    private String content;
    private int likes;
    private List<String> comments;

//  contructor, getters, setters
}
```

Jackson provides us with `ObjectMapper` which you will use for serializing/deserializing. You can see how straightforward it is in the following example:

```java
// Step 1
Post post = new Post(
        1,
        new Date(),
        "I learned how to use jackson!",
        10,
        Arrays.asList("Well done!", "Great job!")
);

// Step 2
ObjectMapper objectMapper = new ObjectMapper();

// Step 3
String postAsString = objectMapper.writeValueAsString(post);

System.out.println(postAsString);
```

In Step 1, you create a post with some data. In Step 2, you create an `ObjectMapper` which provides us with an easy way to convert Java objects to JSON and vice versa. In Step 3, you use the `writeValueAsString` method which generates JSON and returns it as a string.

After running the code in the console, you will get the following result:

```
{"id":1,"createdDate":1655112861424,"content":"I learned how to use jackson!","likes":10,"comments":["Well done!","Great job!"]}
```

It is important to note that we are using the `Date` from `java.util` because Jackson supports it by default, and Jackson will serialize the `Date` to the timestamp format (number of milliseconds since January 1st, 1970, UTC).

You have converted the object to JSON in a few lines of code, but reading it in such a form is quite inconvenient. Let's get a **formatted JSON string**: for this purpose, you will use the `writerWithDefaultPrettyPrinter()` method for constructing a writer that will serialize objects using prettyprinter for indentation.

```java
String postAsString = objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(post);
```

Here is our JSON:

```
{
  "id" : 1,
  "createdDate" : 1654027200000,
  "content" : "I learned how to use jackson!",
  "likes" : 10,
  "comments" : [ "Well done!", "Great job!" ]
}
```

§3. The @JsonProperty annotation

As you see above, Jackson uses field names (`id` , `createdDate` , etc) as a key in JSON. But what if you need to change the key or don't want to put any field in JSON at all? Do you need to completely change the entire class? Jackson developers took care of these situations and provided us with a set of annotations you can use for such cases. Let's look at the most basic of them in this and the next sections.

The first annotation you will talk about is `@JsonProperty`. You can use this annotation in different cases, two of which you'll discuss below.

Firstly, `@JsonProperty` allows you to change the name used in JSON as a key. Let's change the name from `createdDate` to `postedAt`:

```java
public class Post {
    private int id;

    @JsonProperty("postedAt")
    private Date createdDate;

    private String content;
    private int likes;
    private List<String> comments;

}
```

Here is our JSON:

```json
{
  "id" : 1,
  "content" : "I learned how to use jackson!",
  "likes" : 10,
  "comments" : [ "Well done!", "Great job!" ],
  "postedAt" : 1654027200000 // here
}
```

As you can see, now you have `postedAt` instead of `createdDate`. Pretty easy, isn't it?

The second case is to use `@JsonProperty` to denote a non-standard getter/setter that will be used on a JSON property.

For example, if you want to display `createdDate` in a readable format (like `01-01-2000`), you need to put `@JsonProperty` above the method that will do this conversion:

```java
public class Post {
    private int id;
    private Date createdDate;
    private String content;
    private int likes;
    private List<String> comments;


    @JsonProperty("createdDate")

    public String getReadableCreatedDate() {

        return (new SimpleDateFormat("dd-MM-yyyy")).format(createdDate);

    }



// contructor, getters, setters

}
```

Here is our JSON with the formatted date:

```
{
  "createdDate" : "01-06-2022",
  "content" : "I learned how to use jackson!",
  "likes" : 10,
  "comments" : [ "Well done!", "Great job!" ]
}
```

§4. @JsonIgnore

`@JsonIgnore` helps us ignore some Java class fields.

For instance, the message ID (`id`) is sensitive information and you would not want anyone to know it, so it's best to ignore this field.

```
public class Post {
    @JsonIgnore
    private int id;
    private Date createdDate;
    private String content;
    private int likes;
    private List<String> comments;
}
```

Now let's confirm that there is no `id` in JSON:

```
{
  "content" : "I learned how to use jackson!",
  "likes" : 10,
  "comments" : [ "Well done!", "Great job!" ],
  "createdDate" : 1654027200000
}
```

§5. @JsonPropertyOrder

If you look closely at the example with the `@JsonProperty` annotation, you can see that after changing the name from `createdDate` to `postedAt` in JSON, this field is displayed last. Does it mean that the order of the fields is strictly regulated and you must follow it?

By default, the ordering of the fields in the serialized JSON depends on the JDK. They may come in the declaration order but it's not guaranteed.

`@JsonPropertyOrder` allows us to set a specific order when serializing a Java object.

```
@JsonPropertyOrder({
            "likes",
            "comments",
            "createdDate", // here you can also use 'postedAt'
            "content",
    })
public class Post {
    @JsonIgnore
    private int id;

    @JsonProperty("postedAt")
    private Date createdDate;

    private String content;
```

Table of contents:

```
    private int likes;

    private List<String> comments;

}
```

Let's check that the order has changed:

```
{
  "likes" : 10,
  "comments" : [ "Well done!", "Great job!" ],
  "postedAt" : 1654027200000,
  "content" : "I learned how to use jackson!"
}
```

As you can see, you used `createdDate` in `@JsonPropertyOrder` but there wouldn't be a problem if you also used `postedAt`.

§6. Deserialization

Finally, knowing how to convert Java objects to JSON, you also need to understand how to do the reverse conversion. For this, you use the `readValue` method from `ObjectMapper`.

```
String inputJson = "{\"id\":1,\"createdDate\":1654027200000,\"content\":\"I learned how to use jackson!\",\"likes\":10,\"comments\":[\"Well done!\",\"Great job!\"]}\n";

ObjectMapper objectMapper = new ObjectMapper();
Post post = objectMapper.readValue(inputJson, Post.class);
```

Let's dive deeper into it and talk about the restrictions for the class that you will create from the JSON string. For Jackson to be able to convert JSON into a Java object of some class, this class must satisfy several conditions:

- The class must have an empty constructor, and the fields must not be `final`. In this case, Jackson will first create an instance of the class, and then put values into all fields through reflection. Setters are only needed if you want to change the value in the fields — Jackson doesn't use them.

```
public class Post {
    private int id;
    private Date createdDate;
    private String content;
    private int likes;
    private List<String> comments;

    public Post() {
    }


// getters, setters

}
```

- The class must have a constructor with the `@JsonCreator` annotation, and all its parameters must have the `@JsonProperty` annotation, which must necessarily contain the name from JSON.

```java
public class Post {
    private final int id;
    private final Date createdDate;
    private String content;
    private int likes;
    private List<String> comments;

    @JsonCreator
    public Post(
            @JsonProperty("id") int id,
            @JsonProperty("createdDate") Date createdDate,
            @JsonProperty("content") String content,
            @JsonProperty("likes") int likes,
            @JsonProperty("comments") List<String> comments) {
        this.id = id;
        this.createdDate = createdDate;
        this.content = content;
        this.likes = likes;
        this.comments = comments;
    }


// getters
}
```

§7. Conclusion

In this topic, you got acquainted with Jackson and discussed how you can use it to convert Java objects to JSON, and back. You also learned about some of the basic annotations that you will constantly encounter while working with Jackson. Specifically, you have covered the following:

- `ObjectMapper` is a Jackson serializer/deserializer and you must use `writeValueAsString` and `readValue` for these operations respectively;
- How to use a few basic annotations such as `@JsonProperty` , `@JsonIgnore` , and `@JsonPropertyOrder` ;
- How deserialization works and what constraints a class must conform to.

As you can see, Jackson is a very flexible library with extensive support for annotations, some of which you have seen in this topic, but it's just scratching the surface of flexibility you can obtain by using them correctly. Now, let's move on and solve some tasks!

Report a typo

39 users liked this piece of theory. 0 didn't like it. **What about you?**

Start practicing   Verify to skip

This content was created 8 months ago and updated 11 days ago. Share your feedback below in comments to help us improve it!

Comments (6) Useful links (1) Show discussion