Name of the Course : LEARNING JAVA 9 – MODULAR PROGRAMMING

Level : Easy

Tool Stack : Java 9

Problem Statement : we'll learn how to to use Jackson to convert JSON into CSV and vice versa.

1. Introduction

In this short tutorial, we'll see how to use Jackson to convert JSON into CSV and vice versa.

There are alternative libraries available, like the CDL class from org.json, but we'll just focus on the Jackson library here.

After we've looked at our example data structure, we'll use a combination of ObjectMapper and CSVMapper to convert between JSON and CSV.

2. Dependencies

Let’s add the dependency for Jackson CSV data formatter:

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-csv</artifactId>

<version>2.9.8</version>

</dependency>

We can always find the most recent version of this dependency on Maven Central.

We'll also add the dependency for the core Jackson databind:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.8</version>

</dependency>

Again, we can find the most recent version of this dependency on Maven Central.

3. Data Structure

Before we reformat a JSON document to CSV, we need to consider how well our data model will map between the two formats.

So first, let's consider what data the different formats support:

We use JSON to represent a variety of object structures, including ones that contain arrays and nested objects

We use CSV to represent data from a list of objects, with each object from the list appearing on a new line

This means that if our JSON document has an array of objects, we can reformat each object into a new line of our CSV file. So, as an example, let's use a JSON document containing the following list of items from an order:

[ {

"item" : "No. 9 Sprockets",

"quantity" : 12,

"unitPrice" : 1.23

}, {

"item" : "Widget (10mm)",

"quantity" : 4,

"unitPrice" : 3.45

} ]

We'll use the field names from the JSON document as column headers, and reformat it to the following CSV file:

item,quantity,unitPrice

"No. 9 Sprockets",12,1.23

"Widget (10mm)",4,3.45

4. Read JSON and Write CSV

First, we use Jackson's ObjectMapper to read our example JSON document into a tree of JsonNode objects:

JsonNode jsonTree = new ObjectMapper().readTree(new File("src/main/resources/orderLines.json"));

Next, let's create a CsvSchema. This determines the column headers, types, and sequence of columns in the CSV file. To do this, we create a CsvSchema Builder and set the column headers to match the JSON field names:

Builder csvSchemaBuilder = CsvSchema.builder();

JsonNode firstObject = jsonTree.elements().next();

firstObject.fieldNames().forEachRemaining(fieldName -> {csvSchemaBuilder.addColumn(fieldName);} );

CsvSchema csvSchema = csvSchemaBuilder.build().withHeader();

Then, we create a CsvMapper with our CsvSchema, and finally, we write the jsonTree to our CSV file:

CsvMapper csvMapper = new CsvMapper();

csvMapper.writerFor(JsonNode.class)

.with(csvSchema)

.writeValue(new File("src/main/resources/orderLines.csv"), jsonTree);

When we run this sample code, our example JSON document is converted to the expected CSV file.

5. Read CSV and Write JSON

Now, let's use Jackson's CsvMapper to read our CSV file into a List of OrderLine objects. To do this, we first create the OrderLine class as a simple POJO:

public class OrderLine {

private String item;

private int quantity;

private BigDecimal unitPrice;

// Constructors, Getters, Setters and toString

}

We'll use the column headers in the CSV file to define our CsvSchema. Then, we use the CsvMapper to read the data from the CSV into a MappingIterator of OrderLine objects:

CsvSchema orderLineSchema = CsvSchema.emptySchema().withHeader();

CsvMapper csvMapper = new CsvMapper();

MappingIterator<OrderLine> orderLines = csvMapper.readerFor(OrderLine.class)

.with(orderLineSchema)

.readValues(new File("src/main/resources/orderLines.csv"));

Next, we'll use the MappingIterator to get a List of OrderLine objects. Then, we use Jackson's ObjectMapper to write the list out as a JSON document:

new ObjectMapper()

.configure(SerializationFeature.INDENT\_OUTPUT, true)

.writeValue(new File("src/main/resources/orderLinesFromCsv.json"), orderLines.readAll());

When we run this sample code, our example CSV file is converted to the expected JSON document.

6. Configuring the CSV File Format

Let's use some of Jackson's annotations to adjust the format of the CSV file. We'll change the ‘item' column heading to ‘name', the ‘quantity' column heading to ‘count', remove the ‘unitPrice' column, and make ‘count' the first column.

So, our expected CSV file becomes:

count,name

12,"No. 9 Sprockets"

4,"Widget (10mm)"

We'll create a new abstract class to define the required format for the CSV file:

@JsonPropertyOrder({

"count",

"name"

})

public abstract class OrderLineForCsv {

@JsonProperty("name")

private String item;

@JsonProperty("count")

private int quantity;

@JsonIgnore

private BigDecimal unitPrice;

}

Then, we use our OrderLineForCsv class to create a CsvSchema:

CsvMapper csvMapper = new CsvMapper();

CsvSchema csvSchema = csvMapper

.schemaFor(OrderLineForCsv.class)

.withHeader();

We also use the OrderLineForCsv as a Jackson Mixin. This tells Jackson to use the annotations we added to the OrderLineForCsv class when it processes an OrderLine object:

csvMapper.addMixIn(OrderLine.class, OrderLineForCsv.class);

Finally, we use an ObjectMapper to read our JSON document into an OrderLine array, and use our csvMapper to write the this to a CSV file:

OrderLine[] orderLines = new ObjectMapper()

.readValue(new File("src/main/resources/orderLines.json"), OrderLine[].class);

csvMapper.writerFor(OrderLine[].class)

.with(csvSchema)

.writeValue(new File("src/main/resources/orderLinesReformated.csv"), orderLines);

When we run this sample code, our example JSON document is converted to the expected CSV file.

7. Conclusion

In this quick tutorial, we learned how to read and write CSV files using the Jackson data format library. We also looked at a few configuration options that help us get our data looking the way we want.