8 : JSON Processing

IT4206 - Enterprise Application Development

Level II - Semester 4

Overview

- After completing this section, students should be able to generate and parse JSON data via JSON-P's model and Streaming APIs.
- students should be able to describe JSON-P 1.1 features, such as support for JSON Pointer and JSON Patch.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Generate JSON data with the Model API
  - Parse JSON data with the Model API
  - Generate JSON data with the Streaming API
  - Parse JSON data with the Streaming API

List of sub topics

8.1 Generating JSON data with the Model API

8.2 Parsing JSON data with the Model API

8.3 Generating JSON data with the Streaming API

8.4 Parsing JSON data with the Streaming API

- 8.4.1 JSON Pointer
- 8.4.2 JSON Patch

8.1 Generating JSON data with the Model API

JSON-P Model API

- The JSON-P Model API allows us to generate an in-memory representation of a JSON object.
- This API is more flexible than the Streaming API.

8.1 Generating JSON data with the Model API

- JSON-P Model API is the JsonObjectBuilder class.
- It has several overloaded add() methods.
- It can be used to add properties and their corresponding values to generated JSON data.

8.1 Generating JSON data with the Model API

```
package net.ensode.javaee8book.jsonpobject;
//other imports omitted for brevity.
import javax.inject.Named;
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonWriter;
@Named
@SessionScoped
public class JsonpBean implements Serializable{
private String jsonStr;
@Inject
private Customer customer;
public String buildJson() {
JsonObjectBuilder jsonObjectBuilder =
Json.createObjectBuilder();
JsonObject jsonObject = jsonObjectBuilder.
add("firstName", "Scott").
add("lastName", "Gosling").
add("email", "sgosling@example.com").
build();
StringWriter stringWriter = new StringWriter();
try (JsonWriter jsonWriter = Json.createWriter(stringWriter))
{
jsonWriter.writeObject(jsonObject);
}
setJsonStr(stringWriter.toString());
return "display_json";
}
//getters and setters omitted for brevity
}
```

8.1 Generating JSON data with the Model API

- By invoking the add() method on an instance of JsonObjectBuilder, we can add **string** values.
  - First parameter of add() method – property name of the generated Json object
  - Second parameter – value of the said property
- In the above example, invocations to the add() method are chained since return value of the add() method is another instance of JsonObjectBuilder.
- The above example is a CDI-named bean corresponding to a larger JSF application.
- After the properties you want are added, the build() method of JsonObjectBuilder must be invoked. This will run an instance of a class implementing the JsonObject interface.

8.1 Generating JSON data with the Model API

- Suppose that we want to generate a String representation of the JSON object we created.
- So that it can be processed by another service/ process.
- This can be achieved by creating an instance of a class implementing the JsonWriter interface.
- By invoking the static createWriter() method of Json class and passing an instance of StringWriter as its sole parameter we can achieve it.
- If we have an instance of the JsonWriter implementation, then we can invoke its writeObject() method by passing our JsonObject instance as its sole parameter.

8.1 Generating JSON data with the Model API

- Now the StringWriter instance will have the string representation of our JSON object as it's value. As a result, invoking it's toString() method will return a string containing our JSON object. A JSON string as follows will be generated,

```
{"firstName":"Scott","lastName":"Gosling","email":"
sgosling@example.com "}
```

- Even though we added only string objects, we are not limited to this type of value. The reason is JsonObjectBuilder has several overloaded versions of its add() method.

8.1 Generating JSON data with the Model API

- All of the available versions of the **add()** method

add(String **name,** BigDecimal **value)**	Adds a BigDecimal value to our JSON object.
add(String **name,** BigInteger **value)**	Adds a BigInteger value to our JSON object.
add(String **name,** JsonArrayBuilder **value)**	Adds an array to our JSON object. A JsonArrayBuilder implementation allows us to create JSON arrays.
add(String **name,** JsonObjectBuilder **value)**	Adds another JSON object to our original JSON object (property values for JSON objects can be other JSON objects). The added JsonObject implementation is built from the provided JsonObjectBuilder parameter.

8.1 Generating JSON data with the Model API

add(String **name,** JsonValue **value)**	Adds another JSON object to our original JSON object (property values for JSON objects can be other JSON objects).
add(String **name,** String **value)**	Adds a String value to our JSON object.
add(String **name,** boolean **value)**	Adds a boolean value to our JSON object.
add(String **name,** double **value)**	Adds a double value to our JSON object.
add(String **name,** int **value)**	Adds a int value to our JSON object.
add(String **name,** long **value)**	Adds a long value to our JSON object.

8.1 Generating JSON data with the Model API

- The **first parameter** of the add() method corresponds to the **name** of the property in our JSON object.
- The **second parameter** corresponds to the **value** of the property.

8.2 Parsing JSON data with the Model API

```
package net.ensode.javaee8book.jsonpobject;

//other imports omitted
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonWriter;

@Named
@SessionScoped
public class JsonpBean implements Serializable{

  private String jsonStr;

  @Inject
  private Customer customer;


  public String parseJson() {
    JsonObject jsonObject;
    try (JsonReader jsonReader = Json.createReader(
         new StringReader(jsonStr))) {
           jsonObject = jsonReader.readObject();
         }

    customer.setFirstName(
      jsonObject.getString("firstName"));
    customer.setLastName(
      jsonObject.getString("lastName"));
    customer.setEmail(jsonObject.getString("email"));

    return "display_parsed_json";
  }

  //getters and setters omitted

}
```

8.2 Parsing JSON data with the Model API

- A StringReader object must be created in order to parse an existing JSON string. This object will pass the string object containing the JSON to be parsed as a parameter.

- Next the resulting StringReader instance is passed to the static createReader() method of the Json class. This will return an instance of JsonReader.

8.2 Parsing JSON data with the Model API

- By invoking the readObject() method on it, we can obtain an instance of JsonObject.

- In the code example above, to get the values of all properties in our JSON object we used the getString() method. The sole argument is the name of the property we want to retrive. The return value is the value of the property.

8.2 Parsing JSON data with the Model API

- Several other methods to obtain values of other types.

get(Object **key)**	Retrieves an instance of a class implementing the JsonValue interface.
getBoolean(String **name)**	Retrieves a boolean value corresponding to the given key.
getInt(String **name)**	Retrieves a int value corresponding to the given key.
getJsonArray(String **name)**	Retrieves the instance of a class implementing the JsonArray interface corresponding to the given key.

8.2 Parsing JSON data with the Model API

- Several other methods to obtain values of other types.

getJsonNumber(String name)	Retrieves the instance of a class implementing the JsonNumber interface corresponding to the given key.
getJsonObject(String name)	Retrieves the instance of a class implementing the JsonObject interface corresponding to the given key.
getJsonString(String name)	Retrieves the instance of a class implementing the JsonString interface corresponding to the given key.
getString(String name)	Retrieves a String corresponding to the given key.

8.2 Parsing JSON data with the Model API

- The **String parameter** of the method corresponds to the key name.
- The **return value** is the JSON property value we wish to retrieve.

8.3 Generating JSON data with the Streaming API

JSON-P Streaming API

- Allows sequential reading of a JSON object from a stream (a subclass of java.io.OutputStream or a subclass of java.io.Writer).
- It is faster and more memory efficient than the Model API.
- It is more limited, since the JSON data needs to be read sequentially and we cannot access specific JSON properties directly the way the Model API allows.

8.3 Generating JSON data with the Streaming API

- The JSON Streaming API has a **JsonGenerator** class that we can use to generate JSON data and write it to a stream.
- This class has several overloaded **write()** methods, which can be used to add properties and their corresponding values to the generated JSON data.

8.3 Generating JSON data with the Streaming API

```
package net.ensode.javaee8book.jsonpstreaming;

//other imports omitted
import javax.json.Json;
import javax.json.stream.JsonGenerator;
import javax.json.stream.JsonParser;
import javax.json.stream.JsonParser.Event;

@Named
@SessionScoped
public class JsonpBean implements Serializable {

  private String jsonStr;

  @Inject
  private Customer customer;

  public String buildJson() {
    StringWriter stringWriter = new StringWriter();
    try (JsonGenerator jsonGenerator =
      Json.createGenerator(stringWriter)) {
       jsonGenerator.writeStartObject().
         write("firstName", "Larry").
         write("lastName", "Gates").
         write("email", "lgates@example.com").
         writeEnd();
     }

     setJsonStr(stringWriter.toString());
     return "display_json";
 }

 //getters and setters omitted
}
```

8.3 Generating JSON data with the Streaming API

- By invoking the createGenerator() static method of the Json class, an instance of JsonGenerator can be created.
- Two overloaded versions of the above(Provided by JSON-P API) :
  - Takes an instance of a class that extends java.io.Writer(ex: StringWriter)
  - Takes an instance of a class that extends java.io.OutputStream
- First we must invoke the **writeStartObject()** method on JsonGenerator. It will write the Json start object character("{") and returns another instance of JsonGenerator. This will allow us to chain **write()** invocations to add properties to our JSON stream.

8.3 Generating JSON data with the Streaming API

- By using the write() method on Jsongenerator, properties to the JSON stream can be added.
- Two parameters are used.
  - First parameter – a string corresponding to the name of the property we are adding
  - Second parameter – value of the property
- Just as the previous examples we have used only string values but we are not limited to strings.
- The JSON-P streaming API provides several overloaded write() methods.

8.3 Generating JSON data with the Streaming API

- All of the available versions of the **write()** method:

write(String name, BigDecimal value)	Writes a BigDecimal value to our JSON stream.
write(String name, BigInteger value)	Writes a BigInteger value to our JSON stream
write(String name, JsonValue value)	Writes a JSON object to our JSON stream (property values for JSON streams can be other JSON objects)
write(String name, String value)	Writes a String value to our JSON stream
write(String name, boolean value)	Writes a boolean value to our JSON stream
write(String name, double value)	Writes a double value to our JSON stream

8.3 Generating JSON data with the Streaming API

write(String name, int value)	Writes an int value to our JSON stream
write(String name, long value)	Writes a long value to our JSON stream

- The first parameter of the write() method corresponds to the name of the property we are adding to our JSON stream.
- The second parameter corresponds to the value of the property.
- Once we are done adding properties to our JSON stream, we need to invoke the **writeEnd()** method on **JsonGenerator.**

8.3 Generating JSON data with the Streaming API

- This method adds the JSON end object character (represented by a closing curly brace (}) in JSON strings).
- At this point, our stream or reader is populated with the JSON data we generated.
- What we do with it depends on our application logic.
- In our example, we simply invoked the **toString()** method of our **StringReader** to obtain the **String** representation of the JSON data we created.

8.4 Parsing JSON data with the Streaming API

```
package net.ensode.javaee8book.jsonpstreaming;

//other imports omitted
import javax.json.Json;
import javax.json.stream.JsonGenerator;
import javax.json.stream.JsonParser;
import javax.json.stream.JsonParser.Event;

@Named
@SessionScoped
public class JsonpBean implements Serializable {

  private String jsonStr;

  @Inject
  private Customer customer;

  public String parseJson() {

    StringReader stringReader = new StringReader(jsonStr);

    JsonParser jsonParser = Json.createParser(stringReader);

    Map<String, String> keyValueMap = new HashMap<>();
    String key = null;
    String value = null;

    while (jsonParser.hasNext()) {
        JsonParser.Event event = jsonParser.next();

        if (event.equals(Event.KEY_NAME)) {
            key = jsonParser.getString();
        } else if (event.equals(Event.VALUE_STRING)) {
            value = jsonParser.getString();
        }

        keyValueMap.put(key, value);
    }

    customer.setFirstName(keyValueMap.get("firstName"));
    customer.setLastName(keyValueMap.get("lastName"));
    customer.setEmail(keyValueMap.get("email"));

    return "display_parsed_json";
  }

  //getters and setters omitted
```

8.4 Parsing JSON data with the Streaming API

- In order to read JSON data using the Streamin API we must create an instance of JsonParser by invoking the static createJsonParser() method on the Json class.
- There are two overloaded versions of this method :
  - Takes an instance of a class that extends java.io.InputStream
  - Takes an instance of a class that extends java.io.Reader
- java.io.StringReader which is used in the above example is a subclass of the java.io.Reader

8.4 Parsing JSON data with the Streaming API

- Next, to loop through the JSON data we invoke the **hasNext()** method on JsonParser, which will return true if there is more data to be read and false otherwise.
- To obtain the type of data that we have just read we use the JsonParser.next() method which will return an instance of JsonParser.Event.
- In the above example we check only for keynames (i.e firstName, lastName and email) and the corresponding string values. The type of data is checked by comparing the event returned by JsonParser.next() against several values defined in the event enum defined in JsonParser.

8.4 Parsing JSON data with the Streaming API

- All of the possible events that can be returned from **JsonParser.next()**

Event.START_OBJECT	Indicates the **start** of a JSON object.
Event.END_OBJECT	Indicates the **end** of a JSON object.
Event.START_ARRAY	Indicates the start of an array.
Event.END_ARRAY	Indicates the end of an array.
Event.KEY_NAME	Indicates the name of a JSON property was read; we can obtain the key name by invoking getString() on JsonParser.
Event.VALUE_TRUE	Indicates that a Boolean value of **true** was read.
Event.VALUE_FALSE	Indicates that a Boolean value of **false** was read.
Event.VALUE_NULL	Indicates that a **null** value was read.
Event.VALUE_NUMBER	Indicates that a **numeric** value was read
Event.VALUE_STRING	Indicates that a **string** value was read

8.4 Parsing JSON data with the Streaming API

- String values can be retrieved by invoking **getString()** on **JsonParser.**
- Numeric values can be retrieved in several different formats in JsonParser.

getInt()	Retrieves the numeric value as an **int**.
GetLong()	Retrieves the numeric value as a **long**
getBigDecimal()	Retrieves the numeric value as an instance of java.math.BigDecimal

- Using the isIntegralNumber() Method, a value of **true** will be returned if the numeric value can be safely cast to an **int** or a **long**.

8.4.1 JSON pointer

- JSON Pointer is supported by JSON-P 1.1, introduced in Java EE 8. JSON pointer is an Internet Engineering Task Force(IETF) standard that defines a string syntax to identify a specific value within a JSON document.
- Consider the following :

```
{
 "dateOfBirth": "1997-03-03",
 "firstName": "David",
 "lastName": "Heffelfinger",
 "middleName": "Raymond",
 "salutation": "Mr"
}
```

8.4.1 JSON pointer

- To get the value of *lastName* property, the json pointer expression we would use is : **“/lastName”**
- If the JSON document consisted of an array, the property should be pre fixed with the index in the array.
- To obtain the lastName property of the second element in the following JSON array, the JSON pointer expression would be **“/1/lastName”. “/1”** is the element index in the array


```
[
 {
   "dateOfBirth": "1997-01-01",
   "firstName": "David",
   "lastName": "Delabassee",
   "salutation": "Mr"
 },
 {
   "dateOfBirth": "1997-03-03",
   "firstName": "David",
   "lastName": "Heffelfinger",
   "middleName": "Raymond",
   "salutation": "Mr"
 }
]
```

8.4.1 JSON pointer

- The following code depicts how the new JSON-P JSON Pointer API is used to perform the above task

```
package net.ensode.javaee8book.jsonpointer;
//imports omitted

@Path("jsonpointer")
public class JsonPointerDemoService {

  private String jsonString; //initialization omitted

  @GET
  public String jsonPointerDemo() {
    initializeJsonString(); //method body omitted for brevity
    JsonReader jsonReader = Json.createReader
        (new StringReader(jsonString));
    JsonArray jsonArray = jsonReader.readArray();
    JsonPointer jsonPointer = Json.createPointer("/1/lastName");

    return jsonPointer.getValue(jsonArray).toString();
  }
}
```

8.4.1 JSON pointer

- First need to create an instance of **javax.json.JsonReader** by invoking the static **createReader()** method on **javax.json.Json.**
- The **createReader()** method takes an instance of any class implementing the **java.io.Reader** interface as an argument.
- In our example, we are creating a new instance of **java.io.StringReader** on the fly, and passing our JSON string as a parameter to its constructor.

8.4.1 JSON pointer

- *There is an overloaded version of* ***JSON.createReader()*** *that takes an instance of any class implementing* ***java.io.InputStream.***
- JSON document consists of an array of objects.
- So, we populate an instance of javax.json.JsonArray by invoking the readArray() method on the JsonReader object we created (if our JSON document had consisted of a single JSON object, we would have invoked JsonReader.readObject() instead).

8.4.1 JSON pointer

- We have populated our **JsonArray** variable.
- We create an instance of **javax.json.JsonPointer** and initialize it with the JSON Pointer expression we want to use to obtain the value we are searching for.
- Remember that we are looking for the value of the **lastName** property in the second element of the array, therefore, the appropriate JSON Pointer expression is **/1/lastName.**

8.4.1 JSON pointer

- we have created an instance of JsonPointer with the appropriate JSON Pointer expression.
- Simply invoke its **getValue()** method, passing our **JsonArray** object as a parameter.
- Then invoke **toString()** on the result.
- The return value of this invocation will be the value of the **lastName** property on the JSON document.

8.4.2 JSON Patch

- This provides a series of operations that can be applied to a JSON document. Also an IETF standard and introduced by JSON-P 1.1

JSON Patch Operation	Description
Add	Adds an element to a JSON document.
remove	Removes an element from a JSON document.
replace	Replaces a value in a JSON document with a new value
move	Moves a value in a JSON document from its current location in the document to a new position.
copy	Copies a value in a JSON document to a new location in the document.
test	Verifies that the value in a specific location in a JSON document is equal to the specified value.

8.4.2 JSON Patch

- JSON-P supports all of the preceding JSON Patch operations, which rely on JSON Pointer expressions to locate the source and target locations in JSON documents.
- How we can use JSON Patch with JSON-P 1.1 is given below

```
package net.ensode.javaee8book.jsonpatch;

//imports omitted for brevity

@Path("jsonpatch")
public class JsonPatchDemoService {

  private String jsonString;

  @GET
  public Response jsonPatchDemo() {
    initializeJsonString(); //method declaration omitted
    JsonReader jsonReader = Json.createReader(
        new StringReader(jsonString));
    JsonArray jsonArray = jsonReader.readArray();
    JsonPatch jsonPatch = Json.createPatchBuilder()
            .replace("/1/dateOfBirth", "1977-01-01")
            .build();
    JsonArray modifiedJsonArray =jsonPatch.apply(jsonArray);

    return Response.ok(modifiedJsonArray.toString(),
    MediaType.APPLICATION_JSON).build();
  }
}
```

8.4.2 JSON Patch

- In this example has an array of two individual JSON objects, each with a **dateOfBirth** property (among other properties).
- we create an instance of **JsonArray,** as before, then modify the **dateOfBirth** of the second element in the array.
- In order to do this, we create an instance of **javax.json.JsonPatchBuilder** via the static **createPatchBuilder()** method in the **javax.json.Json** class.

8.4.2 JSON Patch

- We are replacing the value of one of the properties with a new value.
- We use the **replace()** method of our **JsonPatch** instance to accomplish this.
- The first argument in the method is a JSON Pointer expression indicating the location of the property we are going to modify.
- The second argument is the new value for the property.
- As its name implies **JsonPatchBuilder** follows the Builder design pattern, meaning that most of its methods return another instance of **JsonPatchBuilder.**

8.4.2 JSON Patch

- This allows us to chain method calls on the resulting instances of **JsonPatchBuilder** (in this example, we are performing only one operation, but this doesn't have to be the case).
- Once we are done specifying the operation(s) to perform on our JSON object, we create an instance of javax.json.JsonPatch by invoking the build() method on JsonPatchBuilder.

8.4.2 JSON Patch

- Once we have created the patch, we apply it to our JSON object (an instance of JsonArray, in our example), by invoking its patch() method, passing the JSON object as a parameter.
- In our example of how to replace the value of a JSON property with another via JSON Patch support in JSON-P 1.1.
- JSON-P supports all operations currently supported by JSON Patch.