DZone

Introduction to JSON With Java

by Justin Albano MVB · **Sep. 11, 18 · Java Zone · Presentation**

Programming plays an indispensable role in software development, but while it is effective at systematically solving problems, it falls short in a few important categories. In the cloud and distributed age of software, data plays a pivotal role in generating revenue and sustaining an effective product. Whether configuration data or data exchanged over a web-based Application Programming Interface (API), having a compact, human-readable data language is essential to maintaining a dynamic system and allowing non-programmer domain experts to understand the system.

While binary-based data can be very condensed, it is unreadable without a proper editor or extensive knowledge. Instead, most systems use a text-based data language, such as eXtensive Markup Language (XML), Yet Another Markup Language (YML), or Javascript Object Notation (JSON). While XML and YML have advantages of their own, JSON has become a front-runner in the realm of configuration and Representational State Transfer (REST) APIs. It combines simplicity with just enough richness to easily express a wide variety of data.

While JSON is simple enough for programmers and non-programmers alike to read and write, as programmers, our systems must be able to produce and consume JSON, which can be far from simple. Fortunately, there exist numerous Java JSON libraries that efficiently tackle this task. In this article, we will cover the basics of JSON, including a basic overview and common use cases, as well as how to serialize Java objects into JSON and how to deserialize JSON into Java objects using the venerable Jackson library. Note that entirety of the code used for this tutorial can be found on Github.

The Basics of JSON

JSON did not start out as a text-based configuration language. Strictly speaking, it is actually Javascript code that represents the fields of an object and can be used by a Javascript compiler to recreate an object. At its most elementary level, a JSON **object** is a set of key-value pairs —

surrounded by braces and with a colon separating the key from the value; the key in this pair is a string —surrounded by quotation marks, and the value is a JSON value. A JSON value can be any of the following:

- another JSON object
- a string
- a number
- the boolean literal `true`
- the boolean literal `false`
- the literal `null`
- an array of JSON values

For example, a simple JSON object, with only string values, could be:

```
{"firstName": "John", "lastName": "Doe"}
```

Likewise, another JSON object can act as a value, as well as a number, either of the boolean literals or a `null` literal, as seen in the following example:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "alias": null,
  "age": 29,
  "isMale": true,
  "address": {
    "street": "100 Elm Way",
    "city": "Foo City",
    "state": "NJ",
    "zipCode": "01234"
  }
}
```

A JSON **array** is a comma-separated list of values surrounded by square brackets. This list does not contain keys; instead, it only contains any number of values, including zero. An array containing zero values is called an **empty array**. For example:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "cars": [
    {"make": "Ford", "model": "F150", "year": 2018},
    {"make": "Subaru", "model": "BRZ", "year": 2016},
  ],
  "children": []
}
```

JSON arrays may be heterogeneous, containing values of mixed types, such as:

```
["hi", {"name": "John Doe"}, null, 125]
```

It is a highly-followed common convention, though, for arrays to contain the same types of data. Likewise, objects in an array conventionally have the same keys and value types, as seen in the previous example. For example, each value in the `cars` array has a key `make` with a string value, a `key` model with a string value, and a key `year` with a numeric value.

It is important to note that the order of key-value pairs in a JSON object and the order of elements in a JSON array do not factor into the equality of two JSON objects or arrays, respectively. For example, the following two JSON objects are equivalent:

```
{"firstName": "John", "lastName": "Doe"}
{"lastName": "Doe", "firstName": "John"}
```

JSON Files

A file containing JSON data uses the `.json` file extension and has a root of either a JSON object or a JSON array. JSON files may not contain more than one root object. If more than one object is desired, the root of the JSON file should be an array containing the desired objects. For example, both of the following are valid JSON files:

```
[
  {"name": "John Doe"},
  {"name": "Jane Doe"}
]
```

```
{
  "name": "John Doe",
  "age": 29
}
```

While it is valid for a JSON file to have an array as its root, it is also common practice for the root be an object with a single key containing an array. This makes the purpose of the array explicit:

```
{
  "accounts": [
    {"name": "John Doe"},
    {"name": "Jane Doe"}
  ]
}
```

Advantages of JSON

JSON has three main advantages: (1) it can be easily read by humans, (2) it can be efficiently parsed by code, and (3) it is expressive. The first advantage is evident by reading a JSON file. Although some files can become large and complex — containing many different objects and arrays — it is the size of these files that is cumbersome, not the language in which they are written. For example, the snippets above can be easily read without requiring context or using a specific set of tools.

Likewise, code can also parse JSON relatively easily; the entirety of the JSON language can be reduced to a set of simple state machines, as described in the ECMA-404 The JSON Data Interchange Standard. We will see in the following sections that this simplicity has translated into

JSON parsing libraries in Java that can quickly and accurately consume JSON strings or files and produce user-defined Java objects.

Lastly, the ease of interpretation does not detract from its expressibility. Nearly any object structure can be recursively reduced to a JSON object or array. Those fields that are difficult to serialize into JSON objects can be stringified and their string value used in place of their object representation. For example, if we wished to store a regular expression as a value, we could do so by simply turning the regular expression into a string.

With an understanding of the basics of JSON, we can now delve into the details of interfacing with JSON snippets and files through Java applications.

Serializing Java Objects to JSON

While the process of tokenizing and parsing JSON strings is tedious and intricate, there are numerous libraries in Java that encapsulate this logic into easy-to-use objects and methods that make interfacing with JSON nearly trivial. One of the most popular among these libraries is FasterXML Jackson.

Jackson centers around a simple concept: the `ObjectMapper`. An `ObjectMapper` is an object that encapsulates the logic for taking a Java object and serializing it to JSON and taking JSON and deserializing it into a specified Java object. In the former case, we can create objects the represent our model and delegate the JSON creation logic to the `ObjectMapper`. Following the previous examples, we can create two classes to represent our model: a person and an address:

```
1  public class Person {
2
3      private String firstName;
4      private String lastName;
5      private String alias;
6      private int age;
7      private boolean isMale;
8      private Address address;
9
10     public Person(String firstName, String lastName, String alias, int age, boolean isMale, Address address) {
11         this.firstName = firstName;
12         this.lastName = lastName;
13         this.alias = alias;
```

```
        this.age = age;
        this.isMale = isMale;
        this.address = address;
    }

    public Person() {}

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getAlias() {
        return alias;
    }

    public void setAlias(String alias) {
        this.alias = alias;
    }

    public int getAge() {
        return age;
```

```
    }

    public void setAge(int age) {
        this.age = age;
    }

    public boolean isMale() {
        return isMale;
    }

    public void setMale(boolean isMale) {
        this.isMale = isMale;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Person [firstName=" + firstName + ", lastName=" + lastName + ", alias=" + alias + ", age=" + age
                + ", isMale=" + isMale + ", address=" + address + "]";
    }
}

public class Address {

    private String street;
    private String city;
```

```
    private String state;
    private String zipCode;

    public Address(String street, String city, String state, String zipCode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
    }

    public Address() {}

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }


    public String getCity() {

        return city;

    }



    public void setCity(String city) {

        this.city = city;

    }
```

```
    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Override
    public String toString() {
        return "Address [street=" + street + ", city=" + city + ", state=" + state + ", zipCode=" + zipCode + "]";
    }
}
```

```
}
```

The `Person` and `Address` classes are both Plain Old Java Objects (POJOs) and do not contain any JSON-specific code. Instead, we can instantiate these classes into objects and pass the instantiated objects to the `writeValueAsString` method of an `ObjectMapper`. This method produces a JSON object—represented as a `String` of the supplied object. For example:

```
Address johnDoeAddress = new Address("100 Elm Way", "Foo City", "NJ", "01234");
Person johnDoe = new Person("John", "Doe", null, 29, true, johnDoeAddress);

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(johnDoe);

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

Executing this code, we obtain the following output:

```
{"firstName":"John","lastName":"Doe","alias":null,"age":29,"address":{"street":"100 Elm Way","city":"Foo City","state":"NJ"
```

If we pretty print (format) this JSON output using the JSON Validator and Formatter, we see that it nearly matches our original example:

```
{
   "firstName":"John",
   "lastName":"Doe",
   "alias":null,
   "age":29,
   "address":{
      "street":"100 Elm Way",
      "city":"Foo City",
      "state":"NJ",
      "zipCode":"01234"
   },
```

```
    "male":true
}
```

By default, Jackson will recursively traverse the fields of the object supplied to the `writeValueAsString` method, using the name of the field as the key and serializing the value into an appropriate JSON value. In the case of `String` fields, the value is a string; for numbers (such as `int`), the result is numeric; for `boolean`s, the result is a boolean; for objects (such as `Address`), this serialization process is recursively repeated, resulting in a JSON object.

Two noticeable differences between our original JSON and the JSON outputted by Jackson are (1) that the order of the key-value pairs is different and (2) there is a `male` key instead of `isMale`. In the first case, Jackson does not necessarily output JSON keys in the same order in which the corresponding fields appear in a Java object. For example, the `male` key is last key in the outputted JSON even though the `isMale` field is before the `address` field in the `Person` object. In the second case, by default, `boolean` flags of the form `isFooBar` result in key-value pairs with a key of the form `fooBar`.

If we desired the key name to be `isMale`, as it was in our original JSON example, we can annotate the getter for the desired field using the `JsonProperty` annotation and explicitly supply a key name:

```
public class Person {

    // ...

    @JsonProperty("isMale")
    public boolean isMale() {
        return isMale;
    }

    // ...
}
```

Rerunning the application results in the following JSON string, which matches our desired output:

```
{
```

```
   "firstName":"John",
   "lastName":"Doe",
   "alias":null,
   "age":29,
   "address":{
      "street":"100 Elm Way",
      "city":"Foo City",
      "state":"NJ",
      "zipCode":"01234"
   },
   "isMale":true
}
```

In cases where we wish to remove `null` values from the JSON output (where a missing key implicitly denotes a `null` value rather than explicitly including the `null` value), we can annotate the affected class as follows:

```
@JsonInclude(Include.NON_NULL)
public class Person {
    // ...
}
```

This results in the following JSON string:

```
{
   "firstName":"John",
   "lastName":"Doe",
   "age":29,
   "address":{
      "street":"100 Elm Way",
      "city":"Foo City",
      "state":"NJ",
```

```
        "zipCode":"01234"
    },
    "isMale":true
}
```

While the resulting JSON string is equivalent to the JSON string that explicitly included the `null` value, the second JSON string saves space. This efficiency is more applicable to larger objects that have many `null` fields.

Serializing Collections

In order to create a JSON array from Java objects, we can pass a `Collection` object (or any subclass) to the `ObjectMapper`. In many cases, a `List` will be used, as it most closely resembles the JSON array abstraction. For example, we can serialize a `List` of `Person` objects as follows:

```
Address johnDoeAddress = new Address("100 Elm Way", "Foo City", "NJ", "01234");
Person johnDoe = new Person("John", "Doe", null, 29, true, johnDoeAddress);

Address janeDoeAddress = new Address("200 Boxer Road", "Bar City", "NJ", "09876");
Person janeDoe = new Person("Jane", "Doe", null, 27, false, janeDoeAddress);

List<Person> people = List.of(johnDoe, janeDoe);

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(people);

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

Executing this code results in the following output:

```
[
    {
        "firstName":"John",
        "lastName":"Doe",
```

```
        "age":29,
        "address":{
           "street":"100 Elm Way",
           "city":"Foo City",
           "state":"NJ",
           "zipCode":"01234"
        },
        "isMale":true
     },
     {
        "firstName":"Jane",
        "lastName":"Doe",
        "age":27,
        "address":{
           "street":"200 Boxer Road",
           "city":"Bar City",
           "state":"NJ",
           "zipCode":"09876"
        },
        "isMale":false
     }
]
```

Writing Serialized Objects to a File

Apart from storing the resulting JSON into a `String` variable, we can write the JSON string directly to an output file using the `writeValue` method, supplying a `File` object that corresponds to the desired output file (such as `john.json`):

```
Address johnDoeAddress = new Address("100 Elm Way", "Foo City", "NJ", "01234");
Person johnDoe = new Person("John", "Doe", null, 29, true, johnDoeAddress);

ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(new File("john.json"), johnDoe);
```

```
5   mapper.writeValue(new File("johnnyjson"), johnDoe);
```

The serialized Java object can also be written to any `OutputStream`, allowing the outputted JSON to be streamed to the desired location (e.g. over a network or on a remote file system).

Deserializing JSON Into Java Objects

With an understanding of how to take existing objects and serialize them into JSON, it is just as important to understand how to deserialize existing the JSON into objects. This process is common when deserializing configuration files and resembles its serializing counterpart, but it also differs in some important ways. First, JSON does not inherently retain type information. Thus, looking at a snippet of JSON, it is impossible to know the type of the object it represents (without explicitly adding an extra key, as is the case with the `_class` key in a MongoDB database). In order to properly deserialize a JSON snippet, we must explicitly inform the deserializer of the expected type. Second, keys present in the JSON snippet may not be present in the deserialized object. For example, if a newer version of an object — containing a new field — has been serialized into JSON, but the JSON is deserialized into an older version of the object.

In order to tackle the first problem, we must explicitly inform the `ObjectMapper` the desired type of the deserialized object. For example, we can supply a JSON string representing a `Person` object and instruct the `ObjectMapper` to deserialize it into a `Person` object:

```
1  String json = "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"alias\":\"Jay\","
2      + "\"age\":29,\"address\":{\"street\":\"100 Elm Way\",\"city\":\"Foo City\","
3      + "\"state\":\"NJ\",\"zipCode\":\"01234\"},\"isMale\":true}";
4
5  ObjectMapper mapper = new ObjectMapper();
6  Person johnDoe = mapper.readValue(json, Person.class);
7  System.out.println(johnDoe);
```

Executing this snippet results in the following output:

```
1  Person [firstName=John, lastName=Doe, alias=Jay, age=29, isMale=true, address=Address [street=100 Elm Way, city=Foo City, s
```

It is important to note that there are two requirements that the deserialized type (or any recursively deserialized types) must have:

1. The class must have a default constructor
2. The class must have setters for each of the fields

Additionally, the `isMale` getter must be decorated with the `JsonProperty` annotation (added in the previous section) denoting that the key name for that field is `isMale`. Removing this annotation will result in an exception during deserialization because the `ObjectMapper`, by default, is configured to fail if a key is found in the deserialized JSON that does not correspond to a field in the supplied type. In particular, since the `ObjectMapper` expects a `male` field (if not configured with the `JsonProperty` annotation), the deserialization fails since there is no field `male` found in the `Person` class.

In order to remedy this, we can annotate the `Person` class with the `JsonIgnoreProperties` annotation. For example:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Person {
    // ...
}
```

We can then add a new field, such as `favoriteColor` in the JSON snippet, and deserialize the JSON snippet into a `Person` object. It is important to note, though, that the value of the `favoriteColor` key will be lost in the deserialized `Person` object since there is no field to store the value:

```
String json = "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"alias\":\"Jay\","
    + "\"age\":29,\"address\":{\"street\":\"100 Elm Way\",\"city\":\"Foo City\","
    + "\"state\":\"NJ\",\"zipCode\":\"01234\"},\"isMale\":true, \"favoriteColor\":\"blue\"}";

ObjectMapper mapper = new ObjectMapper();
Person johnDoe = mapper.readValue(json, Person.class);
System.out.println(johnDoe);
```

Executing this snippet results in the following output:

```
Person [firstName=John, lastName=Doe, alias=Jay, age=29, isMale=true, address=Address [street=100 Elm Way, city=Foo City, s
```

Deserializing Collections

In some cases, the JSON we wish to deserialize has an array as its root value. In order to properly deserialize JSON of this form, we must instruct the `ObjectMapper` to deserialize the supplied JSON snippet into a `Collection` (or a subclass) and then cast the result to a `Collection` object with the proper generic argument. In many cases, a `List` object is used, as in the following code:

```
String json = "[{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":29,"
    + "\"address\":{\"street\":\"100 Elm Way\",\"city\":\"Foo City\","
    + "\"state\":\"NJ\",\"zipCode\":\"01234\"},\"isMale\":true},"
    + "{\"firstName\":\"Jane\",\"lastName\":\"Doe\",\"age\":27,"
    + "\"address\":{\"street\":\"200 Boxer Road\",\"city\":\"Bar City\","
    + "\"state\":\"NJ\",\"zipCode\":\"09876\"},\"isMale\":false}]";

ObjectMapper mapper = new ObjectMapper();
@SuppressWarnings("unchecked")
List<Person> people = (List<Person>) mapper.readValue(json, List.class);

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

Executing this snippet results in the following output:

```
[{firstName=John, lastName=Doe, age=29, address={street=100 Elm Way, city=Foo City, state=NJ, zipCode=01234}, isMale=true},
```

It is important to note that the cast used above is an unchecked cast since the `ObjectMapper` cannot verify that the generic argument of `List` is correct *a priori*; i.e., that the `ObjectMapper` is actually returning a `List` of `Person` objects and not a `List` of some other objects (or not a `List` at all).

Reading JSON From an Input File

Although it is useful to deserialize JSON from a string, it is common that the desired JSON will come from a file rather than a supplied string. In

this case, the `ObjectMapper` can be supplied any `InputStream` or `File` object from which to read. For example, we can create a file named `john.json` on the classpath which contains the following:

```
{"firstName":"John","lastName":"Doe","alias":"Jay","age":29,"address":{"street":"100 Elm Way","city":"Foo City","state":"NJ
```

Then, the following snippet can be used to read from this file:

```
ObjectMapper mapper = new ObjectMapper();
Person johnDoe = mapper.readValue(new File("john.json"), Person.class);
System.out.println(johnDoe);
```

Executing this snippet results in the following output:

```
Person [firstName=John, lastName=Doe, alias=null, age=29, isMale=true, address=Address [street=100 Elm Way, city=Foo City,
```

Conclusion

While programming is a self-evident part of software development, it is not the only skill that developers must possess. In many applications, data drives an application: configuration files create dynamism, and web-service data allows disparate parts of a system to communicate with one another. In particular, JSON has become one of the *de facto* data languages in many Java applications. Learning not only how this language is structured but how to incorporate JSON into Java applications can add an important set of tools to a Java developer's toolbox.

Like This Article? Read More From DZone

Don't Parse, Use Parsing Objects

Maven Skipping Tests

Grails Goodness: Pass JSON Configuration Via Command Line

Free DZone Refcard
Java Containerization

Topics: JAVA , JSON , CONFIGURATION , PARSING , MAVEN

Opinions expressed by DZone contributors are their own.

IN PROGRESS