Atbash JSON Smart

Rudy De Busscher

Version 1.1.1, 04/08/2022

Table of Contents

User manual for Atbash JSON.

Release Notes

1.1.1

1. Capture `NumberFormatException` for ill-formed JSON strings.

1.1.0

1. Support for converting JSON Array to Java `Set`.

1.0.0

Rewrite `accessors-smart` part to remove the dependency on ASM and usage of `setAccessible` so that it runs on newer JDK versions.

Breaking changes

1. Primitives are no longer supported in some cases of getter and setter usage. They are still supported when using public properties in classes.
2. The package of `FieldPropertyNameMapper` is changed.
3. The package of `FieldFilter` is changed.
4. `BeanAccessConfig` is removed and also the possibilities to map Classes and Fields (other alternatives still exist)

0.9.2

1. Support for creating instances of immutable class through the builder pattern.

0.9.1

1. Support for @JsonProperty to define JSON property name
2. SPI for defining JSON property name
3. Integrated access-smart for additional features

Breaking changes

1. Due to integration of *access-smart*, some internal classes are using now other dependent class names (to Atbash package names). But should not affect developer code which only uses @PublicAPI.

0.9.0

1. Initial release, based on the code of JSON smart v2.
2. Support for @MappedBy for easy customized mapping logic.
3. Support for *top-level* Parameterized Types
4. Optimization for use within Octopus JWT

Introduction

JSON Smart v2, is a small fast library for converting POJO to and from JSON in Java 7.

When creating the JSON and JWT support within Octopus, I found out it was difficult to have custom logic for a few cases.

That was the reason to start from the original code and tailor it to my needs (since the library itself is no longer maintained).

Standard usage

```
  T JSONValue.parse("<json>", Class<T>);
```

Converts the "<json>" string to instance of T.

```
  List<Bean> data = (List<Bean>) JSONValue.parse("<json>", new
TypeReference<List<Bean>>(){});
```

Converts the <json> to instance of List containing Bean instances.

Without the TypeReference information, Atbash JSON is only capable of creating a *List* of *JSONObject*.

```
  List<JSONObject> data = JSONValue.parse("<json>", ArrayList.class);
```

```
  String JSONValue.toJSONString(T);
```

Converts the POJO T to a JSON String.

Any method that parses JSON string can throw a `ParseException`. It is a good practice to catch the `AtbashException` at a general location in your program to avoid any exception propagation to the end user.

Customizations

Define JSON property name

By default, the field name is used as property name within JSON.

However, there are scenarios that you need to create a Java class for a JSON (or JWT payload) which needs to comply with some externally defined names. These property names are probably not valid Java property names according to your naming standard.

The solution is to specify the fields with the @JsonProperty (be.atbash.json.annotate.JsonProperty) annotation. The value of this annotation determine the JSON property name. A Java class defined

```
public class Foo {

    @JsonProperty("property")
    private String stringValue;

    private int intValue;

    // Getters and setters for properties
}
```

will result in a JSON like this

```
{"property":"value1","intValue":123}
```

SPI for defining JSON property name

Atbash JSON has also an SPI available to support multiple ways of defining the property name.

Key here is the interface *be.atbash.json.asm.mapper.FieldPropertyNameMapper*. This interface is used to determine the property name for a field.

```
public interface FieldPropertyNameMapper {

    String getPropertyName(Accessor accessor);

}
```

So implement this interface, and define the classname within a service loader definition file (within /META-INF/services/be.atbash.json.asm.mapper.FieldPropertyNameMapper)

When you are not able to determine the property name (based on additional annotations or some general mapping rule based on the field name), just return *null*.

You give then the opportunity that other implementations of this interface come up with a name or that the default (property name = field name) will be used.

Custom JSON creation

The creation of the JSON can be customized in 2 ways. You can implement the interface *be.atbash.json.JSONAware* or define a custom *Writer* with the *@MappedBy* annotation.

```
public interface JSONAware {

    /**
     * @return JSON text
     */
    String toJSONString();

}
```

The result of the *toJSONString()* method will be added to the JSON output. One can make use of the *JSONObject* class to help in the creation of JSON Strings,

```
    public String toJSONString() {
        JSONObject result = new JSONObject();
        result.put("key1", key1);
        result.put("key2", key2);
        result.put("key3", key3);
        for (Map.Entry<String, String> entry : additional.entrySet()) {
            result.put(entry.getKey(), entry.getValue());
        }
        return result.toJSONString();
    }
```

You need to make sure that you serialize the complete object tree to JSON.

Another option, but very similar, is to use an annotation to indicate the code which needs to be called when the Object needs to be Serialized to JSON. This way, the code to create the JSON can be kept out of the class itself.

Annotate the Object with *be.atbash.json.parser.MappedBy* and specify the Writer within the *writer()* member.

```
@MappedBy(writer = PriceJSONWriter.class)
```

and

```
public class PriceJSONWriter implements JSONWriter<PriceWithWriter> {

    @Override
    public <E extends PriceWithWriter> void writeJSONString(E value, Appendable out)
throws IOException {
        out.append(String.format("\"%s%s\"", value.getValue(),
value.getCurrency().toJSONString()));
    }
}
```

In this example, the Currency object implements the *JSONAware* interface.

The last option discussed here, is to register the JSONWriter within the system, as follows

```
    JSONValue.registerWriter(MyColor.class, new MyColorWriter());
```

Then the writer is picked up whenever you ask for converting the MyColor class in this example to JSON.

Custom reading of JSON

The conversion from JSON to an object instance can be customized by encoders which can be defined with *@MappedBy*.

The most generic way is to use an implementation of *be.atbash.json.parser.CustomJSONEncoder*

```
public interface CustomJSONEncoder<T> {

    T parse(Object data);

}
```

The data parameter is most of the time an instance of String, but can be any primitive, JSONArray or JSONObject in case the JSON is malformed or has wrong contents (other contents then expected).

There is a special encoder available, *be.atbash.json.writer.CustomBeanJSONEncoder*, which tries to use the setters if they are available, or call the *setCustomValue()* method otherwise. An example can be seen at the test class *be.atbash.json.testclasses.Token* and *be.atbash.json.testclasses.TokenJSONEncoder*.

An implementation of this interface or the class, needs a no argument constructor.

Both classes needs to be specified by a @MappedBy annotation, *encoder()* member for the simple CustomJSONEncoder implementation, *beanEncoder()* member for CustomBeanJSONEncoder class.

Another customization is possible by registering encoders into the system itself, and then they don't need to be defined by a *mappedBy* annotation. (It has more flexibility but is more difficult)

In the case where the bean is an immutable instance, not default no argument constructor and no setters, there is a specific CustomBeanJSONEncoder available called CustomBeanBuilderJSONEncoder.

Subclasses of CustomBeanBuilderJSONEncoder can also be defined as the value of the *beanEncoder* member of the @MappedBy annotation. Besides the class which will be created, a Bilder class needs to be defined also.

```
   public class ImmutableBeanJSONEncoder extends
CustomBeanBuilderJSONEncoder<ImmutableBean, ImmutableBeanBuilder> {
```

This specific encoder must implements 2 methods

```
    void setBuilderValue(U builder, String key, Object value);

    T build(U builder);
```

The *setBuilderValue* needs to call the corresponding method on the builder for the key property read from the JSON. The *build* method should then create the instance of the requested class, most likely by calling the *build()* method of the builder.

Start by extending the JSONEncoder<T> class and register it by

```
   JSONValue.registerEncoder(<target>.class, new CustomEncoder());

   JSONValue.registerEncoder(new TypeReference<MyType<...>>() {}, new
CustomEncoder());
```

The second statement is for registering a Typed reference. This workaround is required to compensate for the Type erasure which is performed by Java.

After registering, this encoder are used when you ask to *parse* a certain String to the specified type.

The test classes have examples if you want to use this type of customization.

@JsonIgnore

When a field is marked with the @JsonIgnore annotation, it is ignored during the encoding and decoding process.

It can also be used in combination with the @MappedBy.beanEncoder feature. Such a field (which is annotated with @JsonIgnore) will not handled by the default bean encoder when the JSON property key and field name matches, but the value will always be passed to the *setValue()* of the CustomBeanJSONEncoder.