Jakarta JSON Binding Users Guide

Table of Contents

JSON-B is a standard binding layer for converting Java objects to/from JSON messages. It defines a default mapping algorithm for converting existing Java classes to JSON, while enabling developers to customize the mapping process through the use of Java annotations.

Installation

Coordinates

Maven Dependencies

XML

```
<dependencies>
    <!-- JSON-P -->
    <dependency>
        <groupId>org.glassfish</groupId>
        <artifactId>jakarta.json</artifactId>
        <version>1.1.5</version>
        <scope>runtime</scope>
    </dependency>

    <!-- JSON-B API -->
    <dependency>
        <groupId>jakarta.json.bind</groupId>
        <artifactId>jakarta.json.bind-api</artifactId>
        <version>1.0.1</version>
    </dependency>

    <!-- Yasson (JSON-B implementation) -->
    <dependency>
        <groupId>org.eclipse</groupId>
        <artifactId>yasson</artifactId>
        <version>1.0.3</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

JSON Binding API

Default mapping

Default mapping is a set of rules used by JSON-B engine be default without any customization annotations and custom configuration provided.

The main entry point in JSON-B is the Jsonb class. It provides a set of overloaded toJson and fromJson methods to serialize Java objects to JSON documents and deserialize them back. Jsonb instances are thread safe and can be reused. The recommended way is to have a single instance per configuration type.

Default Jsonb engine can be created like this:

JAVA

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
// ...
Jsonb jsonb = JsonbBuilder.create();
```

Mapping an object

The sample below demonstrates serialization and deserialization of a simple POJO.

JAVA

```
public static class Dog {
    public String name;
    public int age;
    public boolean bites;
}

// Create a dog instance
Dog dog = new Dog();
dog.name = "Falco";
dog.age = 4;
dog.bites = false;

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dog);

// Deserialize back
dog = jsonb.fromJson("{\"name\":\"Falco\",\"age\":4,\"bites\":false}", Dog.class);
```

Mapping a collection

JSON-B supports raw collections serialization.

JAVA

```
// List of dogs
List dogs = new ArrayList();
dogs.add(falco);
dogs.add(cassidy);

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dogs);

// We can also deserialize back into a raw collection, but since there is no way to infer a type here,
// the result will be a list of java.util.Map instances with string keys.
dogs = jsonb.fromJson(result, ArrayList.getClass());
```

Mapping a generic collection

JSON-B supports collections and generic collections handling. For proper deserialization the runtime type of resulting object needs to be passed to JSON-B during deserialization. It can be done a way shown below.

JAVA

```
// List of dogs
List<Dog> dogs = new ArrayList<>();
dogs.add(falco);
dogs.add(cassidy);

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dogs);

// Deserialize back
dogs = jsonb.fromJson(result, new ArrayList<Dog>(){}.getClass().getGenericSuperclass());
```

Customized mapping

Your mappings can be customized in many different ways. You can use JSON-B annotations for compile time customizations and JsonbConfig class for runtime customizations.

The sample below shows how to create JSON-B engine with custom configuration:

JAVA

```
// Create custom configuration
JsonbConfig config = new JsonbConfig();

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);
```

Formatted output

JAVA

```
// Create custom configuration with formatted output
JsonbConfig config = new JsonbConfig()
    .withFormatting(true);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

// Use it!
String result = jsonb.toJson(pojo);
```

Changing property names

By default, JSON property name is the same as Java property name. A common use case is serializing properties using different names. This can be achieved using @JsonbProperty annotation on fields or globally using JsonbNamingStrategy.

@JsonbProperty annotation

@JsonbProperty is used to change name of one particular property. It can be placed

- on field, in this case it will affect serialization and deserialization
- on getter, in this case it will affect serialization only
- on setter, in this case it will affect deserialization only

In the sample below property property name will be serialized as 'person-name'.

JAVA

```
public class Person {
    @JsonbProperty("person-name")
    public String name;

    public String profession;
}
```

The resulting JSON document will look like this:

JSON

```
{
    "person-name": "Jason Bourne",
    "profession": "Super Agent"
}
```

The same JSON document will be produced if @JsonbProperty annotation is placed on getter like this:

JAVA

```
public class Person {
    private String name;
    private String profession;

    @JsonbProperty("person-name")
    public String getName() {
        return name;
    }

    public String getProfession() {
        return profession;
    }

    // public setters ...
}
```

This sample demonstrating an ability to write property to one JSON-property and read from another. Property 'name' is serialized to 'name-to-write' property and read from 'name-to-read' property during deserialization.

JAVA

```
public class Person {
    private String name;
    private String profession;

    @JsonbProperty("name-to-write")
    public String getName() {
        return name;
    }

    @JsonbProperty("name-to-read")
    public void setName(String name) {
        this.name = name;
    }

    // other public getters/setters ...
}
```

Naming Strategies

Naming strategy is used to change a default way of constructing property names.

Supported naming strategies are:

- IDENTITY (myMixedCaseProperty)
- LOWER_CASE_WITH_DASHES (my-mixed-case-property)
- LOWER_CASE_WITH_UNDERSCORES (my_mixed_case_property)
- UPPER_CAMEL_CASE (MyMixedCaseProperty)
- UPPER_CAMEL_CASE_WITH_SPACES (My Mixed Case Property)
- CASE_INSENSITIVE (mYmIxEdCaSePrOpErTy)
- Or your custom implementation of JsonbNamingStrategy interface

IDENTITY strategy is the default one.

It can be applied using with withPropertyNamingStrategy method of JsonbConfig class:

JAVA

```
// Custom configuration
JsonbConfig config = new JsonbConfig()
    .withPropertyNamingStrategy(PropertyNamingStrategy.LOWER_CASE_WITH_DASHES);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

...
```

Properties order

To customize the order of serialized properties, JSON Binding provides PropertyOrderStrategy class.

The supported strategies are:

- LEXICOGRAPHICAL (A-Z)
- ANY (order is undefined, in the most cases it will an order in which properties appear in the class)
- REVERSE (Z-A)

Default order strategy is LEXICOGRAPHICAL. Order strategy can be applied globally using withPropertyOrderStrategy method of JsonbConfig class:

JAVA

```
// Custom configuration
JsonbConfig config = new JsonbConfig()
    .withPropertyOrderStrategy(PropertyOrderStrategy.ANY);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

...
```

or using @JsonbPropertyOrder annotation on class:

JAVA

```
@JsonbPropertyOrder(PropertyOrderStrategy.ANY)
public class Person {
    private String name;
    private String profession;

    // public getters/setters ...
}
```

Ignoring properties

By default, JSONB ignores properties with a non public access. All public properties - either public fields or non public fields with public getters are serialized into JSON text.

Excluding properties can be done with a @JsonbTransient annotation. Class properties annotated with @JsonbTransient annotation are ignored by JSON Binding engine. The behavior is different depending on where @JsonbTransient annotation is placed.

- On field: Property is ignored during serialization and deserialization.
- On getter: Property is ignored during serialization only.

- On setter: Property is ignored during deserialization only.

Serialization of this class

JAVA

```
@JsonbPropertyOrder(PropertyOrderStrategy.ANY)
public class Person {
    @JsonbTransient
    private String name;

    private String profession;

    // public getters/setters ...
}
```

will produce the following JSON document:

JSON

```
{
    "profession": "Super Agent"
}
```

If @JsonbTransient annotation is placed on getter like this:

JAVA

```
public class Person {
    private String name;
    private String profession;

    @JsonbTransient
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // other public getters/setters ...
}
```

'name' property won't be serialized, but will be deserialized.

Null handling

By default JSON-B doesn't serialize fields with null values. This may be a not desired behaviour. There are three different ways to change default null handling.

- On a class or package level using @JsonbNillable annotation

JAVA

```
@JsonbNillable
public class Person {
    private String name;
    private String profession;

    // public getters/setters ...
}
```

- On individual properties using @JsonbProperty annotation with nillable=true parameter

JAVA

```
public class Person {
    @JsonbProperty(nillable=true)
    private String name;

    private String profession;

    // public getters/setters ...
}
```

- Globally using withNullValues method of JsonbConfig class

JAVA

```
// Create custom configuration
JsonbConfig nillableConfig = new JsonbConfig()
    .withNullValues(true);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(nillableConfig);

...
```

Custom instantiation

By default, public default no-arguments constructor is required for deserialization. In many scenarios this requirement is too strict. JSON-B provides @JsonbCreator annotation which can be used to annotate a custom constructor with parameters or a static factory method used to create a class instance.

The sample below shows how @JsonbCreator annotation can be used on a custom constructor. @JsonbProperty annotation on the constructor parameter is required for proper parameter substitution. In this case a value of JSON property 'name' will be passed to the constructor.

JAVA

```
public class Person {
    private String name;
    private String profession;

    @JsonbCreator
    public Person(@JsonbProperty("name") String name) {
        this.name = name;
    }

    // public getters/setters ...
}
```

Date/Number Formats

By default JSON-B uses ISO formats to serialize and deserialize date and number fields. Sometimes it's required to override these settings. It can be done using @JsonbDateFormat and @JsonbNumberFormat annotations on fields:

JAVA

```
public class Person {
    private String name;

    @JsonbDateFormat("dd.MM.yyyy")
    private LocalDate birthDate;

    @JsonbNumberFormat("#0.00")
    private BigDecimal salary;

    // public getters/setters ...
}
```

or globally using withDateFormat method of JsonbConfig class:

JAVA

```
// Create custom configuration
JsonbConfig config = new JsonbConfig()
    .withDateFormat("dd.MM.yyyy", null);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

...
```

Binary Encoding

JSON-B supports mapping of binary data. By default it uses BYTE encoding, but it can be easily customized using BinaryDataStrategy class which provides support for the most common binary data encodings:

- BYTE
- BASE_64
- BASE_64_URL

The sample below shows a creation of Jsonb engine with BASE_64_URL encoding:

JAVA

```
// Create custom configuration
JsonbConfig config = new JsonbConfig()
    .withBinaryDataStrategy(BinaryDataStrategy.BASE_64);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

...
```

Adapters

In some cases it may be difficult to serialize/deserialize a class the desired way. Sometimes it's not possible to put custom annotations on the source code because you don't have access to it or custom annotations don't solve the problem. In this case adapters is another option to try.

Adapter is a class implementing javax.json.bind.adapter.JsonbAdapter interface. It has a custom code to convert the "unmappable" type (Original) into another one (Adapted) that is serialized/deserialized the desired way. It's similar to how type adapters in JAXB work.

Let's take a look at the sample. Imagine that you have a Customer object with all customer details. In one scenario you need

to serialize the whole object, in another you need to provide only its id and name. The solution could be to to serialize it as it is for the first scenario and create an adapter to JsonObject which has only data required for the second scenario.

JAVA

```
public class Customer {
    private int id;
    private String name;
    private String organization;
    private String position;

    // public getters/setters ...
}

public class CustomerAnnotated {
    @JsonbProperty("customer_id")
    private int id;

    @JsonbProperty("customer_name")
    private String name;

    // public getters/setters ...
}

public class CustomerAdapter implements JsonbAdapter<Customer, CustomerAnnotated> {
    @Override
    public CustomerAnnotated adaptToJson(Customer c) throws Exception {
        CustomerAnnotated customer = new Customer();
        customer.setId(c.getId());
        customer.setName(c.getName());
        return customer;
    }

    @Override
    public Customer adaptFromJson(CustomerAnnotated adapted) throws Exception {
        Customer customer = new Customer();
        customer.setId(adapted.getId());
        customer.setName(adapted.getName());
        return customer;
    }
}
```

First scenario:

JAVA

```
// Create Jsonb with default configuration
Jsonb jsonb = JsonbBuilder.create();

// Create customer
Customer c = new Customer();

// Initialization code is skipped

// Serialize
jsonb.toJson(customer);
```

Result:

JSON

```
{
    "id": 1,
    "name": "Jason Bourne",
    "organization": "Super Agents",
    "position": "Super Agent"
}
```

Second scenario:

JAVA

```
// Create custom configuration
JsonbConfig config = new JsonbConfig()
    .withAdapters(new CustomerAdapter());

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

// Create customer
Customer c = new Customer();

// Initialization code is skipped

// Serialize
jsonb.toJson(customer);
```

Result:

JSON

```
{
    "customer_id": 1,
    "customer_name": "Jason Bourne",
}
```

Serializers/Deserializers

Sometimes adapters mechanism is not enough and low level access to JSONP parser/generator is needed.

From the spec:

Serializer is a class implementing javax.json.bind.serializers.JsonbSerializer interface. It is used to serialize the type it's registered on (Original). On serializing of Original type JSONB calls JsonbSerializer::serialize method. This method has to contain a custom code to serialize Original type using provided JsonpGenerator.

Deserializer is a class implementing javax.json.bind.serializers.JsonbDeserializer interface. It is used to deserialize the type it's registered on (Original). On deserialization of Original type JSONB calls JsonbDeserializer::deserialize method. This method has to contain a custom code to deserialize Original type using provided JsonpParser.

Let's take a look at the sample. Imagine that we would like to serialize and deserialize a list of pet carriers. A carried pet defined by an abstract class Animal. It can be any of its subclasses. We would like to properly serialize and deserialize it. In order to do it we need to preserve a type information in JSON document and use it on deserialization. It can be done using custom serializer/deserializer pair.

Model:

JAVA

```
public static class Customer {
    private int id;
    private String name;
    private String organization;
    private String position;

    // public getters/setters
}
```

Serializer/Deserializer:

JAVA

```
public static class CustomerSerializer implements JsonbSerializer<Customer> {
    @Override
    public void serialize(Customer customer, JsonGenerator generator, SerializationContext ctx) {
        generator.writeStartObject();
        generator.write("customer_id", customer.getId());
        generator.write("customer_name", customer.getName());
        generator.writeEnd();
    }
}

public static class CustomerDeserializer implements JsonbDeserializer<Customer> {
    @Override
    public Customer deserialize(JsonParser parser, DeserializationContext ctx, Type rtType) {
        Customer customer = new Customer();
        JsonParser.Event next;

        // Moving parser by hand looking for customer_id and customer_name properties
        while ((next = parser.next()) != JsonParser.Event.END_OBJECT) {
            if (next == JsonParser.Event.KEY_NAME) {
                String jsonKeyName = parser.getString();

                // Move to json value
                parser.next();

                if ("customer_id".equals(jsonKeyName)) {
                    customer.setId(parser.getInt());
                } else if ("customer_name".equals(jsonKeyName)) {
                    customer.setName(parser.getString());
                }
            }
        }
        return customer;
    }
}
```

Usage:

JAVA

```
// Create pojo
Customer customer = new Customer();
customer.setId(1);
customer.setName("Freddie");

// Also configurable with @JsonbSerializer / JsonbDeserializer on properties and class.
JsonbConfig config = new JsonbConfig()
        .withSerializers(new CustomerSerializer())
        .withDeserializers(new CustomerDeserializer());

Jsonb jsonb = JsonbBuilder.create(config);
String json = jsonb.toJson(customer);
Customer result = jsonb.fromJson(json, Customer.class);
```

Strict I-JSON support

I-JSON (https://tools.ietf.org/html/draft-ietf-json-i-json-06) ("Internet JSON") is a restricted profile of JSON. JSON-B fully supports I-JSON by default with three exceptions:

- JSON Binding does not restrict the serialization of top-level JSON texts that are neither objects nor arrays. The restriction should happen at application level.
- JSON Binding does not serialize binary data with base64url encoding.
- JSON Binding does not enforce additional restrictions on dates/times/duration.

Full support mode can be switched on like it's shown below:

JAVA

```
// Create custom configuration
JsonbConfig config = new JsonbConfig()
    .withStrictIJSON(true);

// Create Jsonb with custom configuration
Jsonb jsonb = JsonbBuilder.create(config);

...
```

Further Information

- JSON-B official web site: https://eclipse-ee4j.github.io/jsonb-api
- Jakarta JSON Binding: https://projects.eclipse.org/projects/ee4j.jsonb
- Mailing list: jsonb-dev@eclipse.org
- Yasson (Compatible Implementation): https://github.com/eclipse-ee4j/yasson

Last updated 2019-07-02 12:18:41 +0200