Holon Platform JSON Module - Reference manual

Table of Contents

1. Introduction

The Holon Platform **JSON** module provides the support for the JSON data-interchange format, including configuration facilities, data abstraction, PropertyBox type support and seamless integration with the most popular *JSON* mapping and processing libraries: Jackson and Gson.

Through the Json API, the JSON serialization and deserialization using Java object types can be handled in an abstract and implementation-independent way, easily dealing with common JSON mapping concerns such as *temporal* types consistent support and generic collection types, besides a full support of the Holon platform Property model out-of-the-box.

Furthermore, the JSON module faces the following integration concerns:

- *Gson* **GsonBuilder** configuration
- *Jackson* **ObjectMapper** configuration
- Full support of the **Java 8 date and time API** for the `java.time.*` data types serialization and deserialization
- Consistent `Date` data types handling using the **ISO-8601** format

- **JAX-RS** integration and auto-configuration
- **Spring boot** auto-configuration

1.1. Sources and contributions

The Holon Platform **JSON** module source code is available from the GitHub repository https://github.com/holon-platform/holon-json.

See the repository `README` file for information about:

- The source code structure.
- How to build the module artifacts from sources.
- Where to find the code examples.
- How to contribute to the module development.

2. Obtaining the artifacts

The Holon Platform uses Maven for projects build and configuration. All the platform artifacts are published in the **Maven Central Repository**, so there is no need to explicitly declare additional repositories in your project `pom` file.

At the top of each *section* of this documentation you will find the Maven *coordinates* (group id, artifact id and version) to obtain the artifact(s) as a dependency for your project.

A **BOM (Bill Of Materials)** `pom` is provided to import the available dependencies for a specific version in your projects. The Maven coordinates for the core BOM are the following:

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-json-bom</artifactId>
<version>5.2.3</version>
```

The BOM can be imported in a Maven project in the following way:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.json</groupId>
      <artifactId>holon-json-bom</artifactId>
      <version>5.2.3</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

The Holon Platform provides an **overall Maven BOM (Bill of Materials)** to easily obtain all the available platform artifacts.

See Obtain the platform artifacts for details.

3. What's new in version 5.2.x

- Improved nested and hierarchical `PropertyBox` JSON serialization and deserialization support.
- Support for JDK 9+ module system using `Automatic-Module-Name`.
- Support for JAX-RS API version **2.1**.

4. What's new in version 5.1.x

- The new Json API is now available to easily deal with JSON serialization and deserialization in a JSON mapper/parser independent way, with full `PropertyBox` data type support.
- Improved support for *temporal* (date and time) data types, including the `java.time.*` Date and Time API data types. See Date and time data types.
- A more consistent and flexible `Property` set serialization and deserialization strategy when dealing with `PropertyBox` data types, with full support for nested `PropertyBox` data types and a for the property paths naming hierarchy. See `Property` set serialization and deserialization strategies.

5. PropertyBox type handling

The Holon platform PropertyBox API is a versatile and general-purpose data container object, which uses the Property model API to represent a data attribute and to manage the value associated to it.

The JSON module provides configuration facilities to transparently and consistently deal with the `PropertyBox` data type when using one of the currently supported JSON mapper implementations.

From a general point of view, the PropertyBox data type handling is implemented according to the following strategies:

5.1. Serialization

A PropertyBox instance is serialized as a JSON object. Each Property of the PropertyBox property set is serialized as a JSON object *field*, using the property **name** as JSON field name and the property **value** provided by the PropertyBox instance as JSON field value.

See the Property documentation section for information about the Holon platform Property model and PropertyBox API.

The property value is serialized according to its **type**, following the serialization strategies of the concrete JSON mapper that is used. Nested PropertyBox data types are fully supported and the standard *dot* (.) notation can be used, besides the more formal property path hierarchy definition, to define the JSON object internal structure with nested objects support. See Property set serialization and deserialization strategies for details.

Using standard JSON objects favors portability, allowing to parse and map PropertyBox representations in a language independent way.

5.1.1. PropertyBox properties serialization strategy

By default, only the Path type properties of the PropertyBox property set are taken into account for PropertyBox JSON serialization, and the Path name is used as the property serialization **name**.

See the PathProperty documentation section for details about Path type properties.

This behaviour can be changed to include any Property of the PropertyBox property set in JSON serialization, using one of the serialization strategies listed in the PropertyBoxSerializationMode enumeration, through the PROPERTYBOX_SERIALIZATION_MODE configuration property, available from the JsonConfigProperties interface.

The PROPERTYBOX_SERIALIZATION_MODE configuration property value can be provided either using the PropertyBox property set *configuration* attributes container or as a concrete JSON mapper implementation configuration attribute. See the currently supported JSON mapper implementations documentation to learn about this second configuration option.

```
StringProperty NAME = StringProperty.create("name"); ①
VirtualProperty<String> VRT = VirtualProperty.create(String.class, pb -> "(" + pb
.getValue(NAME) + ")")
    .name("vrt"); ②

final PropertySet<?> PROPERTY_SET = PropertySet.builderOf(NAME, VRT).
withConfiguration(
    JsonConfigProperties.PROPERTYBOX_SERIALIZATION_MODE, PropertyBoxSerializationMode
.ALL).build(); ③

PropertyBox propertyBox = PropertyBox.builder(PROPERTY_SET).set(NAME, "test").build();
④
```

① Declare a standard `String` type `PathProperty`

② Declare a `VirtualProperty` which provides the `NAME` property value within brackets, setting `vrt` as property name

③ Configure the `PropertyBox` serialization mode using the `PropertySet` configuration to include **all** the property set properties (not only the `Path` type ones according to the default behaviour)

④ When the `PropertyBox` is serialized into JSON, also the `VRT` property will be included in the JSON object, using `vrt` as serialization name

5.2. Deserialization

A `PropertyBox` instance is deserialized as a JSON object. Each JSON object *field* is deserialized as a `Property` of the `PropertyBox` property set, matching its name with the property name, and setting the property value in the `PropertyBox` instance with the deserialized JSON object *field* value.

See the Property documentation section for information about the Holon platform `Property` model and `PropertyBox` API.

The property value is deserialized according to its **type**, following the deserialization strategies of the concrete JSON mapper that is used. Nested `PropertyBox` data types are fully supported and the standard *dot* (`.`) notation can be used, besides the more formal property path hierarchy definition, to parse the JSON object internal structure and map it to the `PropertyBox` property set. See `Property` set serialization and deserialization strategies for details.

Since a `PropertyBox` type structure is defined through a **property set**, which declares the properties managed by the `PropertyBox` instance, the property set with which to deserialize the JSON object must be provided at JSON deserialization time.

This can be done using the Holon platform Context architecture, providing the deserialization property set as a **context resource** (typically thread-bound). To bind a `PropertySet` instance to the default thread-scoped context resource set, the `execute(Callable operation)` method can be used.

```
final PathProperty<Long> KEY = create("key", long.class);
final PathProperty<String> NAME = create("name", String.class);
final PropertySet<?> PROPERTIES = PropertySet.of(KEY, NAME);

PropertyBox box = PROPERTIES.execute(() -> deserializePropertyBox()); ①
```

① The `deserializePropertyBox()` method, which performs `PropertyBox` JSON deserialization, is executed after binding the `PROPERTIES` as a thread-scoped context resource

When using the Json API, the deserialization *property set* is directly provided to the `Json` API deserialization methods, without the need to use a context resource reference.

5.3. `Property` set serialization and deserialization strategies

When a `PropertyBox` is serialized into JSON and deserialized from JSON, its property set is used to determine the mapping between the JSON object structure and the property set properties, according to the following strategy:

Serialization:: . Any `PropertyBox` type `Property` is serialized as a nested JSON object (using the property name as the field name of the parent JSON object). . If a `Path` type property declares a **parent** paths hierarchy, it is serialized in a corresponding JSON object hierarchy, using the parent paths to reproduce a consistent nested JSON objects hierarchy. . If the *dot* (`.`) notation is used in `Property` name, the property name is turned into a paths hierarchy using the `.` character as hierarchy separator. Then the `Property` is serialized according to the previous point.

Example of nested PropertyBox serialization

```
final NumericProperty<Long> KEY = NumericProperty.longType("key");

final StringProperty NAME = StringProperty.create("name");
final StringProperty SURNAME = StringProperty.create("surname");

final PropertyBoxProperty NESTED = PropertyBoxProperty.create("nested", NAME, SURNAME
); ①

final PropertySet<?> PROPERTY_SET = PropertySet.of(KEY, NESTED); ②

PropertyBox value = PropertyBox.builder(PROPERTY_SET).set(KEY, 1L)
    .set(NESTED, PropertyBox.builder(NAME, SURNAME).set(NAME, "John").set(SURNAME,
"Doe").build()).build(); ③
```

① `PropertyBox` type property: the property set (`NAME`, `SURNAME`) is declared at creation time

② The actual property set is composed of the `KEY` and the `NESTED` properties

③ The result of the serialized `PropertyBox` value will be: `{"key":1,"nested":{"name":"John","surname":"Doe"}}`

Example of nested Property path serialization

```
final NumericProperty<Long> KEY = NumericProperty.longType("key");
final StringProperty NAME = StringProperty.create("nested.name"); ①
final StringProperty SURNAME = StringProperty.create("nested.surname"); ②

final PropertySet<?> PROPERTY_SET = PropertySet.of(KEY, NAME, SURNAME);

PropertyBox value = PropertyBox.builder(PROPERTY_SET).set(KEY, 1L).set(NAME, "John")
.set(SURNAME, "Doe")
    .build(); ③
```

① The `NAME` property path is defined as a path hierarchy using the *dot* notation: `nested.name`

② The `SURNAME` property path is defined as a path hierarchy using the *dot* notation: `nested.surname`

③ The result of the serialized `PropertyBox` value will be: `{"key":1,"nested":{"name":"John","surname":"Doe"}}`

Deserialization:: . A nested JSON object is deserialized as a `PropertyBox` type `Property`, if it is available in the `PropertyBox` property set with a matching name and path hierarchy. . A nested JSON object field is deserialized in a `Property` with a matching name and path hierarchy, if it is available in the `PropertyBox` property set.

So, for example, the JSON value used in the previous examples:

```
{
  "key":1,
  "nested":{
    "name":"John",
    "surname":"Doe"
  }
}
```

Can be deserialized as a `PropertyBox` indifferently using any of the two property sets shown in the previous examples.

6. Date and time data types

6.1. `java.time.*` data types support

The support for the Java 8 date and time API data types is enabled out-of-the-box, and allows to deal transparently with the temporal data types such as `LocalDate`, `LocalTime` and `LocalDateTime`.

This includes any temporal type `Property` serialized and deserialized within a `PropertyBox`.

See specific implementation documentation for details: Jackson implementation and Gson implementation.

6.2. `java.util.Date` serialization and deserialization

By default, the `java.util.Date` data types are serialized in JSON using the **ISO-8601** format, to provide a standard and more readable way to represent date and time types in the serialized JSON output.

This includes any temporal type `Property` serialized and deserialized within a `PropertyBox`.

See specific implementation documentation for details: Jackson implementation and Gson implementation.

7. The Json API

The Json interface represents a simple Java API to serialize and deserialize Objects to and from JSON, acting as an abstraction layer for a concrete JSON parser implementation.

The `Json` API provides methods to serialize an Object to a JSON representation and to deserialize back an Object from a JSON representation, easily dealing with generic and collection types.

7.1. Obtain an implementation

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-json</artifactId>
<version>5.2.3</version>
```

The JsonProvider interface is used to provide a concrete `Json` API implementation.

A new `JsonProvider` can be registered using the default Java `ServiceLoader` extension, providing a `com.holonplatform.json.JsonProvider` file under the `META-INF/services` classpath folder. The `javax.annotation.Priority` annotation (where less priority value means higher priority order) can be used to order the Json providers when more than one is available.

Thanks to the Java `ServiceLoader` API, each implementation is automatically registered when the corresponding artifact is available in classpath (`holon-jackson` for the **Jackson** implementation and `holon-gson` for the **Gson** one).

To obtain the currently available `Json` implementation, the `get()` or `require()` methods of the `Json` API interface can be used. These methods adopt the following strategy to obtain the current `Json` API implementation:

- If a `Json` implementation is available as a Holon `Context` resource using the `com.holonplatform.json.Json` resource name, this one is returned (See Context documentation for information about context scopes and resources).
- Otherwise, if a `JsonProvider` is registered using the Java `ServiceLoader` API, it is invoked to obtain

the corresponding `Json` implementation. When more than one `JsonProvider` is available, the one with the higher priority is used.

```
Optional<Json> jsonAPI = Json.get(); ①

Json jsonAPI_ = Json.require(); ②
```

① Try to obtain the current `Json` API implementation

② Obtain the current `Json` API implementation, throwing an exception if none available

7.1.1. Currently available implementations

The Holon platform currentlty makes available two `Json` API implementations out-of-the-box:

- A Jackson implementation: see Jackson `Json` API implementation.
- A Gson implementation: see Gson `Json` API implementation.

7.2. Serialization

To serialize an Object in JSON, the `toJson` method is provided:

```
JsonWriter toJson(Object value);
```

This method returns a JsonWriter instance, which represents the JSON serialization result and makes available methods to obtain the JSON data in different formats.

You can use the `JsonWriter` API to obtain the JSON result in the following ways:

- Obtain it as a String
- Obtain it as an array of bytes
- Write it into an `Appendable` writer
- Write it into an `OutputStream` writer, specifying the *charset* or using the default `UTF-8` charset

The `toJsonString` convenience method can be used to serialize the Object and directly obtain the JSON result as a String.

```
Json json = Json.require(); ①

Object myObject = getObject(); // the object to serialize

JsonWriter result = json.toJson(myObject); ②

String asString = result.asString(); ③
byte[] asBytes = result.asBytes(); ④
result.write(new StringWriter()); ⑤
result.write(new ByteArrayOutputStream()); ⑥
result.write(new ByteArrayOutputStream(), StandardCharsets.ISO_8859_1); ⑦

asString = json.toJsonString(myObject); ⑧
```

① Obtain a `Json` implementation

② Serialize an Object

③ Get the JSON result as a String

④ Get the JSON result as an array of bytes

⑤ Write the JSON result into a StringWriter

⑥ Write the JSON result into a ByteArrayOutputStream

⑦ Write the JSON result into a ByteArrayOutputStream using the ISO LATIN-1 charset

⑧ Serialize the Object and obtain the JSON result as a String

The supported data types and their serialization strategies depends on the concrete JSON mapper implementation, including any custom serializer or serialization configuration attribute. The `Json` API simply delegates to the concrete backing implementation the actual JSON serialization operations, seamlessy inheriting its serialization strategy and data types support.

When the `Json` API is obtained using a `JsonProvider`, a default configuration of the backing implementation is provided by the `JsonProvider` implementation itself. To have more control on the concrete implementation configuration and to fine tune the JSON provider setup, you can either create your own `JsonProvider` implementation or directly obtain the `Json` API from the concrete providers implementations. See each available implementation documentation for details.

7.2.1. `PropertyBox` serialization

The Holon platform `PropertyBox` type serialization is fully supported out-of-the-box. The `PropertyBox` serialization strategy is described in the PropertyBox type handling section.

```
Json json = Json.require(); ①

final PathProperty<Long> KEY = create("key", long.class);
final PathProperty<String> NAME = create("name", String.class);
final PropertySet<?> PROPERTIES = PropertySet.of(KEY, NAME);

PropertyBox propertyBox = PropertyBox.builder(PROPERTIES).set(KEY, 1L).set(NAME, "
Test").build(); ②

final StringBuilder sb = new StringBuilder();
json.toJson(propertyBox).write(sb); ③
```

① Obtain the `Json` implementation

② Build a `PropertyBox` with given `PROPERTIES` property set and set the property values

③ Serialize the `PropertyBox` instance and write the JSON result into a `StringBuilder`

7.2.2. Generic Collections serialization

The `Json` API provides some helpful methods to deal with *generic* `Collection` of values, in order to serialize the collection of values into a JSON array.

To serialize a collection of values into a JSON array, the `toJsonArray` method can be used:

```
<T> JsonWriter toJsonArray(Class<T> type, Collection<T> values);
```

The concrete value type has to be provided besides the actual `Collection` value.

The `Json` API provides some overloaded and convenience methods to use an array of values instead of a `Collection` and to directly obtain the JSON result as a String.

```
Json json = Json.require(); ①

Collection<Integer> values = Arrays.asList(1, 2, 3, 4);

JsonWriter result = json.toJsonArray(Integer.class, values); ②
String asString = json.toJsonArrayString(Integer.class, values); ③
result = json.toJsonArray(Integer.class, 1, 2, 3, 4); ④
```

① Obtain the `Json` implementation

② Serialize the collection of `Integer` values

③ Serialize the collection of `Integer` values and obtain the JSON result as a String

④ Serialize an array of `Integer` values

The `PropertyBox` type is fully supported also when serializing to a JSON array. For example, to serialize two `PropertyBox` instances named `box1` and `box2`, simply call the `toJsonArray` method providing the `PropertyBox.class` value type: `json.toJsonArray(PropertyBox.class, box1, box2)`.

7.3. Deserialization

To serialize an Object from JSON, the `fromJson` method is provided:

```
<T> T fromJson(JsonReader reader, Class<T> type);
```

The deserialization method, besides the JSON source, requires the Object `type` into which to deserialize the JSON value, and returns the deserialized instance of the specified type.

The JSON source is provided using the JsonReader API, which makes available a set of methods to obtain the JSON data from different sources:

- From a generic `java.io.Reader`
- From a `String`
- From an array of bytes
- From an `InputStream`, optionally specifying the encoding charset or assuming `UTF-8` by default

```
Json json = Json.require(); ①

MyObject result = json.fromJson(JsonReader.from("[JSON string]"), MyObject.class); ②
result = json.fromJson("[JSON string]", MyObject.class); ③
result = json.fromJson(JsonReader.from(new byte[] { 1, 2, 3 }), MyObject.class); ④
result = json.fromJson(JsonReader.from(new StringReader("[JSON string]")), MyObject
.class); ⑤
result = json.fromJson(JsonReader.from(new ByteArrayInputStream(new byte[] { 1, 2, 3
})), MyObject.class); ⑥
```

① Obtain the `Json` implementation

② Deserialize an object of type `MyObject` using a String as JSON data source

③ Convenience method to provide directly a String as JSON data source

④ Deserialize an object of type `MyObject` using an array of bytes as JSON data source

⑤ Deserialize an object of type `MyObject` using a `Reader` as JSON data source

⑥ Deserialize an object of type `MyObject` using a `InputStream` as JSON data source

The supported data types and their deserialization strategies depends on the concrete JSON mapper implementation, including any custom deserializer or deserialization configuration attribute. The `Json` API simply delegates to the concrete backing implementation the actual JSON deserialization operations, seamlessy inheriting its deserialization strategy and data types support.

When the `Json` API is obtained using a `JsonProvider`, a default configuration of the backing implementation is provided by the `JsonProvider` implementation itself. To have more control on the concrete implementation configuration and to fine tune the JSON provider setup, you can either create your own `JsonProvider` implementation or directly obtain the `Json` API from the concrete providers implementations. See each available implementation documentation for details.

7.3.1. `PropertyBox` deserialization

The Holon platform `PropertyBox` type deserialization is fully supported out-of-the-box. The `PropertyBox` serialization strategy is described in the PropertyBox type handling section.

The `Json` API provides a set of convenience methods to deserialize a `PropertyBox` or a collection of `PropertyBox`. These methods require the deserialization **property set** to be provided. Overloaded methods versions are available to provide, for example, the property set as an `Iterable` (which includes the `PropertySet` interface) or as an array of `Property`.

Some examples:

```
Json json = Json.require(); ①

final PathProperty<Long> KEY = create("key", long.class);
final PathProperty<String> NAME = create("name", String.class);
final PropertySet<?> PROPERTIES = PropertySet.of(KEY, NAME);

PropertyBox result = json.fromJson(JsonReader.from("[JSON string]"), PROPERTIES); ②
result = json.fromJson("[JSON string]", PROPERTIES); ③
result = json.fromJson(JsonReader.from("[JSON string]"), KEY, NAME); ④

List<PropertyBox> results = json.fromJsonArray("[JSON string]", PROPERTIES); ⑤
```

① Obtain the `Json` implementation

② Deserialize a `PropertyBox` instance from a JSON String, using `PROPERTIES` as property set

③ Deserialize a `PropertyBox` instance from a JSON String, using `PROPERTIES` as property set

④ Deserialize a `PropertyBox` instance from a JSON String, using `KEY` and `NAME` properties as property set

⑤ Deserialize a `List` of `PropertyBox` from a JSON array, using `PROPERTIES` as property set

7.3.2. JSON arrays deserialization

The `Json` API provides a set of convenience methods to deserialize a collection of objects from a JSON array. For this purpose, the `fromJsonArray` method can be used:

```
<T> List<T> fromJsonArray(JsonReader reader, Class<T> type);
```

This method behaves in the same way as the `fromJson` method, but returns a `List` of the serialized objects.

8. Supported JSON libraries

The Holon platform JSON module currently supports the Jackson and Gson libraries out-of-the-box.

See the next sections for details about each implementation.

9. Jackson integration

The Jackson library support is provided by the `holon-jackson` artifact:

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-jackson</artifactId>
<version>5.2.3</version>
```

9.1. Jackson `ObjectMapper` configuration

The `JacksonConfiguration` interface can be used to configure a Jackson `ObjectMapper` (or obtain an already configured one) and enable the Holon platform JSON support features, including:

- The `PropertyBox` type support in JSON serialization and deserialization.
- The `java.time.*` temporal types support.
- The `java.util.Date` type handling using the ISO-8601 format

The `JacksonConfiguration` interface makes available methods to configure an existing `ObjectMapper` instance or to obtain an already configured one:

```
ObjectMapper mapper = JacksonConfiguration.configure(new ObjectMapper()); ①

mapper = JacksonConfiguration.mapper(); ②
```

① Configure an `ObjectMapper` instance

② Create and configure a new `ObjectMapper` instance

See the next sections for details about each feature.

9.1.1. `PropertyBox` type support

The `PropertyBoxModule` Jackson Module is provided to configure `PropertyBox` type mapping support for a Jackson `ObjectMapper`, providing a suitable `PropertyBox` type JSON serializer and deserializer. This module is automatically registered in an `ObjectMapper` configured through the `JacksonConfiguration` API, as described in Jackson `ObjectMapper` configuration.

The `PropertyBox` serialization and deserialization strategy follows the rules described in the PropertyBox type handling section.

With a properly configured `ObjectMapper` instance, you can deal with `PropertyBox` type serialization and deserialization just like any another supported object type.

```
final static PathProperty<Long> ID = PathProperty.create("id", Long.class);
final static PathProperty<String> DESCRIPTION = PathProperty.create("description",
String.class);

final static PropertySet<?> PROPERTY_SET = PropertySet.of(ID, DESCRIPTION);

public void serializeAndDeserialize() throws JsonProcessingException {
  ObjectMapper mapper = JacksonConfiguration.mapper(); ①

  PropertyBox box = PropertyBox.builder(PROPERTY_SET).set(ID, 1L).set(DESCRIPTION,
"Test").build(); ②

  // serialize
  String json = mapper.writer().writeValueAsString(box); ③
  // deserialize
  box = PROPERTY_SET.execute(() -> mapper.reader().forType(PropertyBox.class)
.readValue(json)); ④
}
```

① Obtain a properly configured `ObjectMapper` instance

② Build a `PropertyBox` using `PROPERTY_SET` as property set

③ Serialize the `PropertyBox` to JSON.

④ Deserialize back the JSON value into a `PropertyBox` instance using `PROPERTY_SET` as property set, declaring it as thread-bound *Context* resource through the `execute(…)` method

In the example above, the `PropertyBox` instance will be serialized as a JSON object like this:

```
{
  "id": 1,
  "description": "Test"
}
```

9.1.2. `PropertyBox` serialization configuration

As described in the `PropertyBox` properties serialization strategy section, the `PropertyBox` properties serialization strategy can be configured using the `PROPERTYBOX_SERIALIZATION_MODE` configuration property, available from the `JsonConfigProperties` interface.

Besides using the `PropertySet Configuration` container to configure the `PropertyBox` properties serialization strategy, the serialization mode can be configured globally for a Jackson `ObjectMapper` using a Jackson Deserialization context attribute: `PROPERTYBOX_SERIALIZATION_MODE_ATTRIBUTE_NAME`. SInce it is a deserialization context attribute, it has to be configured at `ObjectWriter` level.

```
final ObjectMapper mapper = JacksonConfiguration.mapper();

final ObjectWriter writer = mapper.writer() ①
    .withAttribute(JsonConfigProperties.PROPERTYBOX_SERIALIZATION_MODE_ATTRIBUTE_NAME,
        PropertyBoxSerializationMode.ALL); ②
```

① Obtain an `ObjectWriter`

② Set the `PROPERTYBOX_SERIALIZATION_MODE_ATTRIBUTE_NAME` attribute to `PropertyBoxSerializationMode.ALL` to include all the properties of the `PropertyBox` property set when it is serialized to JSON

9.1.3. Jackson `java.time.*` data types support

The Java 8 date and time API data types serialization and deserialization are supported out-of-the-box when using a Jackson `ObjectMapper` obtained or configured through the `JacksonConfiguration` API (see Jackson `ObjectMapper` configuration).

The `jackson-datatype-jsr310` dependency is included in the artifact dependencies and the default `com.fasterxml.jackson.datatype.jsr310.JavaTimeModule` Jackson module is registered to enable the `java.time.*` objects serialization support.

By default the Jackson `ObjectMapper` is configured to **not write dates as timestamps**, to make the serialized JSON more readable and easy to understand for temporal types.

This way, you can deal transparently with the `java.time.*` serialization and deserialization without the need for further configurations.

Example of a `LocalDate` object serialization and deserialization:

```
ObjectMapper mapper = JacksonConfiguration.mapper(); ①

LocalDate date = LocalDate.of(2018, Month.JANUARY, 5);

String serialized = mapper.writeValueAsString(date); ②

LocalDate deserialized = mapper.readValue(serialized, LocalDate.class); ③
```

① Use a properly configured `ObjectMapper` instance

② Serialize given `LocalDate` in JSON. This will result in `"2018-01-05"`

③ Deserialize back the JSON date into a `LocalDate`

9.1.4. ISO-8601 `java.util.Date` serialization

When using a Jackson `ObjectMapper` obtained or configured through the `JacksonConfiguration` API (see Jackson `ObjectMapper` configuration), the ISO8601DateModule module is registered by default.

This module automatically enables the `java.util.Date` serialization in the **ISO-8601** format, to provide a standard and more readable way to represent date and time types in the serialized JSON output.

So a `java.util.Date` type value is serialized with the following pattern:

```
2018-01-05T10:30:25
```

The timezone offset is included if available.

Since a `java.util.Date` always contains all the date and time parts, the value is serialized including the time part by default.

If the `java.util.Date` value is serialized as a `Property` value within a `PropertyBox`, the `PropertyConfiguration` is checked to obtain the `TemporalType` of the property. If the property `TemporalType` is available, the `java.util.Date` value is serialized according to the property temporal type, i.e. as a `DATE` (only the date part), as a `TIME` (only the time part) or as a `DATE_TIME` (both the date and the time part).

See the PropertyConfiguration documentation for further information.

```
final ObjectMapper mapper = JacksonConfiguration.mapper();

final TemporalProperty<Date> DATE = TemporalProperty.date("date").temporalType
(TemporalType.DATE); ①

Calendar c = Calendar.getInstance();
c.set(2018, 0, 5);

PropertyBox value = PropertyBox.builder(DATE).set(DATE, c.getTime()).build(); ②

String json = JacksonConfiguration.mapper().writeValueAsString(value); ③
```

① Declare a `TemporalProperty` of `java.util.Date` type and set `DATE` as property temporal type

② Use the property within a `PropertyBox`

③ The serialized JSON value will be: `{"date":"2018-01-05"}` not including the time part

When the `java.util.Date` value is not bound to a `Property`, you can use a `ThreadLocal` variable to set the current `TemporalType` which has to be used to serialize the date/time value. The value serialization temporal type can be setted and cleared using the `setCurrentTemporalType(TemporalType temporalType)` and `removeCurrentTemporalType()` methods of the CurrentSerializationTemporalType class.

```
final ObjectMapper mapper = JacksonConfiguration.mapper(); ①

Calendar c = Calendar.getInstance();
c.set(2018, 0, 5);
final Date date = c.getTime();

try {
  CurrentSerializationTemporalType.setCurrentTemporalType(TemporalType.DATE); ②
  String json = mapper.writeValueAsString(date); ③
} finally {
  CurrentSerializationTemporalType.removeCurrentTemporalType(); ④
}
```

① Use a properly configured `ObjectMapper` instance

② Set the current `TemporalType` to `DATE`

③ Only the date part will be serialized in JSON: `"2018-01-05"`

④ Clear the current `TemporalType`

9.2. Jackson `Json` API implementation

Jackson can be used as Json API implementation.

When the `holon-jackson` artifact is present in classpath, a suitable `JsonProvider` is automatically registered, and the `Json` API implementation can be obtained through the `Json.get()` and `Json.require()` methods.

Otherwise, the Jackson `Json` API implementation can be directly obtained using the JacksonJson interface, through one of the `create()` methods.

```
Json jsonApi = Json.require(); ①

jsonApi = JacksonJson.create(); ②
```

① Get the Jackson `Json` API implementation using the registered provider

② Obtain the Jackson `Json` API implementation directly

9.3. JAX-RS integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-jackson-jaxrs</artifactId>
<version>5.2.3</version>
```

A set of JAX-RS extension features are provided to configure a JAX-RS context when using Jackson as JSON provider, to enable all the features provided by the Holon platform JSON support module and to configure the Jackson JAX-RS `ObjectMapper` as described in the Jackson `ObjectMapper` configuration section.

To setup the *Jackson* JAX-RS extensions, the JacksonFeature feature has to be registered in the JAX-RS application.

If you use Jersey or Resteasy as JAX-RS implementation, there is no need to explicitly register the `JacksonFeature`, just ensure the `holon-jackson-jaxrs` jar is in classpath and the *Jackson* support will be **configured automatically**, leveraging on Jersey *AutoDiscoverable* and Resteasy Java Service extensions features.

When the feature is registered and enabled, the following extensions will be available:

- A JAX-RS `MessageBodyReader` and `MessageBodyWriter` for the `application/json` media type to handle the `PropertyBox` type and perform JSON serialization and deserialization, according to the default strategy as described in the PropertyBox type handling section. See `PropertyBox` type deserialization for details about the `PropertyBox` property set handling.
- A JAX-RS `javax.ws.rs.ext.ContextResolver` to provide the Jackson `ObjectMapper` instance to be used for JSON mapping operations. By default, the provided `ObjectMapper` is configured as described in the Jackson `ObjectMapper` configuration section.

9.3.1. `PropertyBox` type deserialization

When a `PropertyBox` is used as a JAX-RS resource method **parameter** (for methods which declare to consume `application/json` media type), the deserialization of the JSON input value into a `PropertyBox` instance needs to know the `PropertySet` to use in order to create the `PropertyBox` instance. For this purpose, the PropertySetRef annotation can be used at method parameter level to declare the `PropertySet`.

The `@PropertySetRef` annotation allows to declare the `PropertySet` instance as the `public static` **field** of a given class, which must be specified in the `value()` annotation attribute. If more than one `public static` field of `PropertySet` type is present in the declared class, the `field()` annotation attribute can be used to specify the field name to use.

```
final static PathProperty<Integer> CODE = PathProperty.create("code", Integer.class);
final static PathProperty<String> NAME = PathProperty.create("name", String.class);

final static PropertySet<?> PROPERTYSET = PropertySet.of(CODE, NAME);

// JAX-RS example endpoint
@Path("test")
public static class Endpoint {

  @PUT
  @Path("serialize")
  @Consumes(MediaType.APPLICATION_JSON)
  public Response create(@PropertySetRef(value = ExampleJackson.class, field =
"PROPERTYSET") PropertyBox data) { ①
    return Response.accepted().build();
  }

  @GET
  @Path("deserialize")
  @Produces(MediaType.APPLICATION_JSON)
  public PropertyBox getData() {
    return PropertyBox.builder(PROPERTYSET).set(CODE, 1).set(NAME, "Test").build();
  }

}

public void jaxrs() {
  Client client = ClientBuilder.newClient(); ②

  PropertyBox box1 = PropertyBox.builder(PROPERTYSET).set(CODE, 1).set(NAME, "Test")
.build();

  client.target("https://host/test/serialize").request().put(Entity.entity(box1,
MediaType.APPLICATION_JSON)); ③

  PropertyBox box2 = PROPERTYSET
      .execute(() -> client.target("https://host/test/deserialize").request().get
(PropertyBox.class)); ④

}
```

① The `data` input parameter is annotated with `@PropertySetRef` to declare the `PropertyBox` deserialization property set

② Create a JAX-RS `Client`

③ Perform a `PUT` request providing a `PropertyBox` value as JSON. At the endpoint resource level, the `PropertyBox` type input parameter of the `serialize` method is annotated with `@PropertySetRef` in order to declare the property set to use to deserialize the property box from JSON

④ Perform a `GET` request for a JSON serialized `PropertyBox` value, providing the `PropertySet` to use

for deserialization as a `Context` thread-bound resource

9.3.2. Deal with the JAX-RS context `ObjectMapper` instance

When using the JAX-RS `JacksonFeature`, a default `javax.ws.rs.ext.ContextResolver` is registered to provide the Jackson `ObjectMapper` instance to be used for JSON mapping operations. The context resolver provides by default an `ObjectMapper` instance configured according to the Jackson `ObjectMapper` configuration of the Holon platform JSON module.

You can replace the default `ObjectMapper` instance of the JAX-RS context in the following ways:

1. Provide a custom `ContextResolver`:

A custom `javax.ws.rs.ext.ContextResolver` for the `ObjectMapper` type can be provided and registered in the JAX-RS application.

To ensure the default `ContextResolver` will not be taken into account, you can use the JAX-RS application configuration property `holon.jackson.disable-resolver`, setting it to `true` to disable the default context resolver.

To ensure `PropertyBox` type JSON serialization and deserialization consistency, the `ObjectMapper` instance should be configured registering the PropertyBoxModule Jackson module. The JacksonConfiguration API can be used for this purpose.

```
@Produces(MediaType.APPLICATION_JSON) ①
public static class MyObjectMapperResolver implements ContextResolver<ObjectMapper> {

  private final ObjectMapper mapper;

  public MyObjectMapperResolver() {
    super();
    mapper = JacksonConfiguration.mapper(); ②
    // additional ObjectMapper configuration
    // ...
  }

  @Override
  public ObjectMapper getContext(Class<?> type) {
    return mapper;
  }

}
```

① The JAX-RS `ContextResolver` instance has to be annotated so that to declare that is bound to the `application/json` media type

② The `JacksonConfiguration` can be used to configure the `ObjectMapper` instance to support the Holon platform JSON features, such as `PropertyBox` type mapping

2. Use the Holon platform `Context`:

The default `ObjectMapper` context resolver looks up for a *context* resource of `ObjectMapper` type using the `ObjectMapper` class name as resource key before returning the default `ObjectMapper` instance.

If the `ObjectMapper` type context resource is found, it is returned and used as JAX-RS `ObjectMapper` instance.

See Context for information about the Holon platform context and context resources handling.

This way, you can provide your own `ObjectMapper` instance using the Holon platform `Context` API to register your `ObjectMapper` instance as a context resource with the appropriate resource key.

```
final ObjectMapper mapper = JacksonConfiguration.mapper();
// additional ObjectMapper configuration
// ...

Context.get().classLoaderScope().map(s -> s.put(ObjectMapper.class.getName(), mapper)
); ①
```

① The *classloader* context scope is used to register a custom `ObjectMapper` instance using the `ObjectMapper` class name as resource key

To ensure `PropertyBox` type JSON serialization and deserialization consistency, the `ObjectMapper` instance should be configured registering the PropertyBoxModule Jackson module. The JacksonConfiguration API can be used for this purpose.

9.3.3. JAX-RS integration configuration

The following JAX-RS application configuration properties are available to tune or disable the Jackson JAX-RS integration features:

- `holon.jackson.disable-resolver`: If this property is present in JAX-RS application properties, the Jackson `ObjectMapper` context resolver auto-configuration is disabled.
- `holon.jackson.disable-autoconfig`: If this property is present in JAX-RS application properties, all the Holon platform Jackson JAX-RS extension features will be disabled.
- `holon.jaxrs.json.pretty-print`: If `true`, enables *pretty printing* of serialized JSON.

9.4. Spring integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-jackson-spring</artifactId>
<version>5.2.3</version>
```

The SpringJacksonConfiguration interface can be used to configure a Spring `RestTemplate`, ensuring

that a `MappingJackson2HttpMessageConverter` is registered and bound to a `ObjectMapper` instance correctly configured for Holon platform Jackson extensions, as described in the Jackson `ObjectMapper` configuration section.

```
@Configuration
class Config {

  @Bean
  public RestTemplate restTemplate() {
    return SpringJacksonConfiguration.configure(new RestTemplate()); ①
  }

}
```

① Create a new `RestTemplate` instance and configure it with the Holon platform JSON support extensions

9.5. Spring Boot integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-jackson-spring</artifactId>
<version>5.2.3</version>
```

The JacksonAutoConfiguration Spring Boot *auto-configuration* class is provided to automatically configure an `ObjectMapper` *bean*, if available in the Spring context, with the Holon platform JSON support extensions, as described in the Jackson `ObjectMapper` configuration section.

This way, the `RestTemplate` instances obtained through the `RestTemplateBuilder` Spring Boot builder will be automatically pre-configured with the Holon platform Jackson extensions.

To disable this auto-configuration feature, the `JacksonAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={JacksonAutoConfiguration.class})
```

10. Gson integration

The Gson library support is provided by the `holon-gson` artifact:

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-gson</artifactId>
<version>5.2.3</version>
```

10.1. Gson `GsonBuilder` configuration

The GsonConfiguration interface can be used to configure a Gson `GsonBuilder` (or obtain an already configured one) and enable the Holon platform JSON support features, including:

- The `PropertyBox` type support in JSON serialization and deserialization.
- The `java.time.*` temporal types support.
- The `java.util.Date` type handling using the ISO-8601 format

The `GsonConfiguration` interface makes available methods to configure an existing `GsonBuilder` instance or to obtain an already configured one:

```
GsonBuilder builder = GsonConfiguration.builder(); ①

builder = GsonConfiguration.configure(new GsonBuilder()); ②
```

① Get a new pre-configured `GsonBuilder`

② Configure a `GsonBuilder` instance

See the next sections for details about each feature.

10.1.1. `PropertyBox` type support

When the `GsonConfiguration` API is used to configure a `GsonBuilder`, serializers and deserializers for the `PropertyBox` type are registered and enabled.

The `PropertyBox` serialization and deserialization strategy follows the rules described in the PropertyBox type handling section.

With a properly configured `GsonBuilder` instance, you can deal with `PropertyBox` type serialization and deserialization just like any another supported object type.

```
final static PathProperty<Long> ID = PathProperty.create("id", Long.class);
final static PathProperty<String> DESCRIPTION = PathProperty.create("description",
String.class);

final static PropertySet<?> PROPERTY_SET = PropertySet.of(ID, DESCRIPTION);

public void serializeAndDeserialize() {
  Gson gson = GsonConfiguration.builder().create(); ①

  PropertyBox box = PropertyBox.builder(PROPERTY_SET).set(ID, 1L).set(DESCRIPTION,
"Test").build(); ②

  // serialize
  String json = gson.toJson(box); ③
  // deserialize
  box = PROPERTY_SET.execute(() -> gson.fromJson(json, PropertyBox.class)); ④
}
```

① Obtain a pre-configured `Gson` instance

② Build a `PropertyBox` using `PROPERTY_SET` as property set

③ Serialize the `PropertyBox` to JSON.

④ Deserialize back the JSON definition to a `PropertyBox` instance using `PROPERTY_SET` as property set, declaring it as thread-bound *Context* resource through the `execute(…)` method

In the example above, the `PropertyBox` instance will be serialized as a JSON object like this:

```
{
  "id": 1,
  "description": "Test"
}
```

10.1.2. `PropertyBox` serialization configuration

As described in the `PropertyBox` properties serialization strategy section, the `PropertyBox` properties serialization strategy can be configured using the `PROPERTYBOX_SERIALIZATION_MODE` configuration property, available from the JsonConfigProperties interface.

Besides using the `PropertySet Configuration` container to configure the `PropertyBox` properties serialization strategy, the serialization mode can be configured globally for a Gson `GsonBuilder` using the `GsonConfiguration` API.

```
GsonBuilder builder = GsonConfiguration.builder(PropertyBoxSerializationMode.ALL); ①

builder = GsonConfiguration.configure(new GsonBuilder(), PropertyBoxSerializationMode
.ALL); ②
```

① Get a new pre-configured `GsonBuilder` and set the serialization mode to `PropertyBoxSerializationMode.ALL` to include all the properties of the `PropertyBox` property set when it is serialized to JSON

② Configure a `GsonBuilder` instance and set the serialization mode to `PropertyBoxSerializationMode.ALL`

10.2. Gson `java.time.*` data types support

The Java 8 date and time API data types serialization and deserialization are supported out-of-the-box when using a `GsonBuilder` obtained or configured through the GsonConfiguration API.

Supported `java.time.*` data types are: `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, `OffsetDateTime`, `ZonedDateTime` and `Instant`.

This way, you can deal transparently with the `java.time.*` serialization and deserialization without the need for further configurations.

Example of a `LocalDate` object serialization and deserialization:

```
Gson gson = GsonConfiguration.builder().create(); ①

LocalDate date = LocalDate.of(2018, Month.JANUARY, 5);

String serialized = gson.toJson(date); ②

LocalDate deserialized = gson.fromJson(serialized, LocalDate.class); ③
```

① Use a properly configured `Gson` instance

② Serialize given `LocalDate` in JSON. This will result in `"2018-01-05"`

③ Deserialize back the JSON date into a `LocalDate`

10.3. ISO-8601 `java.util.Date` serialization

When using a Gson `GsonBuilder` obtained or configured through the GsonConfiguration API, the serialization of the `java.util.Date` type is made using the **ISO-8601** format by default, to provide a standard and more readable way to represent date and time types in the serialized JSON output.

So a `java.util.Date` type value is serialized with the following pattern:

```
2018-01-05T10:30:25
```

The timezone offset is included if available.

Since a `java.util.Date` always contains all the date and time parts, the value is serialized including the time part by default.

If the java.util.Date value is serialized as a Property value within a PropertyBox, the PropertyConfiguration is checked to obtain the TemporalType of the property. If available, the java.util.Date value is serialized according to the property temporal type, i.e. as a DATE (only the date part), as a TIME (only the time part) or as a DATE_TIME (both the date and the time part).

See the PropertyConfiguration documentation for further information.

When the java.util.Date value is not bound to a Property, you can use a ThreadLocal variable to set the current TemporalType which has to be used to serialize the date/time value. The value serialization temporal type can be setted and cleared using the setCurrentTemporalType(TemporalType temporalType) and removeCurrentTemporalType() methods of the CurrentSerializationTemporalType class.

```
Gson gson = GsonConfiguration.builder().create(); ①

Calendar c = Calendar.getInstance();
c.set(2018, 0, 5);
final Date date = c.getTime();

try {
  CurrentSerializationTemporalType.setCurrentTemporalType(TemporalType.DATE); ②
  String json = gson.toJson(date); ③
} finally {
  CurrentSerializationTemporalType.removeCurrentTemporalType(); ④
}
```

① Use a properly configured Gson instance

② Set the current TemporalType to DATE

③ Only the date part will be serialized in JSON: "2018-01-05"

④ Clear the current TemporalType

10.4. Gson Json API implementation

Gson can be used as Json API implementation.

When the holon-gson artifact is present in classpath, a suitable JsonProvider is automatically registered, and the Json API implementation can be obtained through the Json.get() and Json.require() methods.

Otherwise, the Gson Json API implementation can be directly obtained using the GsonJson interface, through one of the create() methods.

```
Json jsonApi = Json.require(); ①

jsonApi = GsonJson.create(); ②
```

① Get the Gson Json API implementation using the registered provider

② Obtain the Gson `Json` API implementation directly

10.5. JAX-RS integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-gson-jaxrs</artifactId>
<version>5.2.3</version>
```

A set of JAX-RS extension features are provided to configure a JAX-RS context when using Gson as JSON provider, to enable all the features provided by the Holon platform JSON support module and to configure the Gson JAX-RS context `Gson` instance, obtained from a `GsonBuilder` configured as described in the Gson `GsonBuilder` configuration section.

To setup the *Gson* JAX-RS extensions, the GsonFeature feature has to be registered in the JAX-RS application.

If you use Jersey or Resteasy as JAX-RS implementation, there is no need to explicitly register the `GsonFeature`, just ensure the `holon-gson-jaxrs` jar is in classpath and the *Gson* support will be **configured automatically**, leveraging on Jersey *AutoDiscoverable* and Resteasy Java Service extensions features.

When the feature is registered and enabled, the following extensions will be available:

- A JAX-RS `MessageBodyReader` and `MessageBodyWriter` for the `application/json` media type to handle the `PropertyBox` type and perform JSON serialization and deserialization, according to the default strategy as described in the PropertyBox type handling section. See `PropertyBox` type deserialization for details about the `PropertyBox` property set handling.
- A JAX-RS `javax.ws.rs.ext.ContextResolver` to provide the `Gson` instance to be used for JSON mapping operations. By default, the provided `Gson` instance is obtained from a `GsonBuilder` configured as described in the Gson `GsonBuilder` configuration section.

10.5.1. `PropertyBox` type deserialization

When a `PropertyBox` is used as a JAX-RS resource method **parameter** (for methods which declare to consume `application/json` media type), the JSON deserialization of the input into a `PropertyBox` instance needs to know the `PropertySet` to use in order to create the property box. For this purpose, the `@PropertySetRef` annotation can be used at method parameter level to declare the `PropertySet` instance to use to deserialize the property box.

The PropertySetRef annotation allows to declare the `PropertySet` instance as the `public static` **field** of a given class, which must be specified in the `value()` annotation attribute. If more than one `public static` field of `PropertySet` type is present in declared class, the `field()` annotation attribute can be used to specify the right field name.

```
final static PathProperty<Integer> CODE = PathProperty.create("code", Integer.class);
final static PathProperty<String> NAME = PathProperty.create("name", String.class);

final static PropertySet<?> PROPERTYSET = PropertySet.of(CODE, NAME);

// JAX-RS example endpoint
@Path("test")
public static class Endpoint {

  @PUT
  @Path("serialize")
  @Consumes(MediaType.APPLICATION_JSON)
  public Response create(@PropertySetRef(value = ExampleGson.class, field =
"PROPERTYSET") PropertyBox data) {
    return Response.accepted().build();
  }

  @GET
  @Path("deserialize")
  @Produces(MediaType.APPLICATION_JSON)
  public PropertyBox getData() {
    return PropertyBox.builder(PROPERTYSET).set(CODE, 1).set(NAME, "Test").build();
  }

}

public void jaxrs() {
  Client client = ClientBuilder.newClient(); ①

  PropertyBox box1 = PropertyBox.builder(PROPERTYSET).set(CODE, 1).set(NAME, "Test")
.build();

  client.target("https://host/test/serialize").request().put(Entity.entity(box1,
MediaType.APPLICATION_JSON)); ②

  PropertyBox box2 = PROPERTYSET
      .execute(() -> client.target("https://host/test/deserialize").request().get
(PropertyBox.class)); ③

}
```

① Create a JAX-RS `Client`

② Perform a `PUT` request providing a `PropertyBox` value as JSON. At the endpoint resource level, the `PropertyBox` type input parameter of the `serialize` method is annotated with `@PropertySetRef` in order to declare the property set to use to deserialize the property box from JSON

③ Perform a `GET` request for a JSON serialized `PropertyBox` value, providing the `PropertySet` to use for deserialization as a `Context` thread-bound resource

10.5.2. Deal with the JAX-RS context `Gson` instance

When using the JAX-RS `GsonFeature`, a default `javax.ws.rs.ext.ContextResolver` is registered to provide the `Gson` instance to be used for JSON mapping operations. The context resolver provides by default an `Gson` instance obtained using a `GsonBuilder` configured according to the Gson `GsonBuilder` configuration of the Holon platform JSON module.

You can replace the default `Gson` instance of the JAX-RS context in the following ways:

1. Provide a custom `ContextResolver`:

A custom `javax.ws.rs.ext.ContextResolver` for the `Gson` type can be provided and registered in the JAX-RS application.

To ensure the default `ContextResolver` will not be taken into account, you can use the JAX-RS application configuration property `holon.gson.disable-resolver`, setting it to `true` to disable the default context resolver.

To ensure `PropertyBox` type JSON serialization and deserialization consistency, the `GsonBuilder` from which the `Gson` instance is obtained should be configured using the GsonConfiguration API.

```
@Produces(MediaType.APPLICATION_JSON) ①
public static class MyObjectMapperResolver implements ContextResolver<Gson> {

  private final Gson gson;

  public MyObjectMapperResolver() {
    super();
    GsonBuilder builder = GsonConfiguration.builder(); ②
    // additional GsonBuilder configuration
    // ...
    gson = builder.create();
  }

  @Override
  public Gson getContext(Class<?> type) {
    return gson;
  }

}
```

① The JAX-RS `ContextResolver` instance has to be annotated so that to declare that is bound to the `application/json` media type

② The `GsonConfiguration` can be used to configure the `GsonBuilder` instance to support the Holon platform JSON features, such as `PropertyBox` type mapping

2. Use the Holon platform `Context`:

The default `Gson` context resolver looks up for a *context* resource of `Gson` type using the `Gson` class name as resource key before returning the default `Gson` instance.

If the `Gson` type context resource is found, it is returned and used as JAX-RS `Gson` instance.

See Context for information about the Holon platform context and context resources handling.

This way, you can provide your own `Gson` instance using the Holon platform `Context` API to register your `Gson` instance as a context resource with the appropriate resource key.

```
GsonBuilder builder = GsonConfiguration.builder();
// additional GsonBuilder configuration
// ...
final Gson gson = builder.create();

Context.get().classLoaderScope().map(s -> s.put(Gson.class.getName(), gson)); ①
```

① The *classloader* context scope is used to register a custom `Gson` instance using the `Gson` class name as resource key

To ensure `PropertyBox` type JSON serialization and deserialization consistency, the `GsonBuilder` from which the `Gson` instance is obtained should be configured using the GsonConfiguration API.

10.5.3. JAX-RS integration configuration

The following JAX-RS application configuration properties are available to tune or disable the Gson JAX-RS integration features:

- `holon.gson.disable-resolver`: If this property is present in JAX-RS application properties, the Gson `Gson` context resolver auto-configuration is disabled.
- `holon.gson.disable-autoconfig`: If this property is present in JAX-RS application propertiesall the Holon platform Gson JAX-RS extension features will be disabled.
- `holon.jaxrs.json.pretty-print`: If `true`, enables *pretty printing* of serialized JSON.

10.6. Spring integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-gson-spring</artifactId>
<version>5.2.3</version>
```

The SpringGsonConfiguration utility interface can be used to configure a Spring `RestTemplate`, ensuring that a `GsonHttpMessageConverter` is registered and bound to a `Gson` instance correctly

configured for Holon platform Gson extensions, as described in the Gson `GsonBuilder` configuration section.

```
@Configuration
class Config {

  @Bean
  public RestTemplate restTemplate() {
    return SpringGsonConfiguration.configure(new RestTemplate()); ①
  }

}
```

① Create a new `RestTemplate` instance and configure it with the Holon platform JSON support extensions

10.7. Spring Boot integration

Maven coordinates:

```
<groupId>com.holon-platform.json</groupId>
<artifactId>holon-gson-spring</artifactId>
<version>5.2.3</version>
```

The GsonAutoConfiguration Spring Boot *auto-configuration* class is provided to automatically configure a `Gson` type **singleton bean**, with the Holon platform JSON support extensions, as described in the Gson `GsonBuilder` configuration section.

This way, the `RestTemplate` instances obtained through the `RestTemplateBuilder` Spring Boot builder will be automatically pre-configured pre-configured with the Holon platform Gson extensions.

The `Gson` bean auto-configuration is triggered only if a `Gson` type bean is not already registered in the Spring context.

To disable this auto-configuration feature, the `GsonAutoConfiguration` class can be excluded:

```
@EnableAutoConfiguration(exclude={GsonAutoConfiguration.class})
```

11. Loggers

By default, the Holon platform uses the SLF4J API for logging. The use of SLF4J is optional: it is enabled when the presence of SLF4J is detected in the classpath. Otherwise, logging will fall back to JUL (`java.util.logging`).

The logger names for the **JSON** module are:

- `com.holonplatform.json.gson` for the *Gson* integration classes
- `com.holonplatform.json.jackson` for the *Jackson* integration classes

12. System requirements

12.1. Java

The Holon Platform JSON module requires Java **8** or higher.