**Marshalling** 🡪

It converts a Java object into XML or JSON(Serialization).

**Unmarshalling** 🡪

It converts XML or JSON into a Java Object(Deserialization).

**Dependency**

**jackson-databind**

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.13.0</version>

</dependency>

This dependency will also transitively add the following libraries to the classpath:

1. jackson-annotations
2. jackson-core

# Understanding the Jackson ObjectMapper class

## **Jackson General Annotations**

### **@JsonProperty**

We can add the @JsonProperty annotation to indicate the property name in JSON.

@JsonProperty is to serialize/deserialize the property name when we're dealing with non-standard getters and setters.

public class MyBean {

public int id;

private String name;

@JsonProperty("name")

public void setTheName(String name) {

this.name = name;

}

@JsonProperty("name")

public String getTheName() {

return name;

}

}

@Test

public void whenUsingJsonProperty\_thenCorrect()

throws IOException {

MyBean bean = new MyBean(1, "My bean");

String result = new ObjectMapper().writeValueAsString(bean);

assertThat(result, containsString("My bean"));

assertThat(result, containsString("1"));

MyBean resultBean = new ObjectMapper()

.readerFor(MyBean.class)

.readValue(result);

assertEquals("My bean", resultBean.getTheName());

}

## **@JsonFormat**

The *@JsonFormat* annotation specifies a format when serializing Date/Time values.

public class EventWithFormat {

public String name;

@JsonFormat(

shape = JsonFormat.Shape.STRING,

pattern = "dd-MM-yyyy hh:mm:ss.SSSZ", locale = "en\_GB")

public Date eventDate;

}

Default output is "createdDate":1482047026009

Output after format "createdDate":"2016-12-18@07:53:34.740+0000"

Using @JsonFormat with shape set to JsonFormat.Shape.NUMBER results in the default output for Date types — as the number of seconds since the epoch.

The parameter pattern is not applicable to this case.

# **Jackson Serialization Annotations**

## **@JsonAnyGetter**

The @JsonAnyGetter annotation allows for the flexibility of using a Map field as standard properties.

public class ExtendableBean {

public String name;

private Map<String, String> properties;

@JsonAnyGetter

public Map<String, String> getProperties() {

return properties;

}

}

Output:

{

"name":"My bean",

"attr2":"val2",

"attr1":"val1"

}

@Test

public void whenSerializingUsingJsonAnyGetter\_thenCorrect()

throws JsonProcessingException

{

public static void main(String[] args) throws J sonMappingException, JsonProcessingException

{

ExtendableBean bean = new ExtendableBean("My bean");

HashMap<String, String> hashMap = new HashMap<>();

hashMap.put("attr1", "val1");

hashMap.put("attr2", "val2");

bean.setProperties(hashMap);

String result = new ObjectMapper().writeValueAsString(bean);

System.out.println(result);

}

}

We can also use the optional argument enabled as false to disable @JsonAnyGetter().

In this case, the Map will be converted as JSON and will appear under the properties variable after serialization.

public class ExtendableBean {

public String name;

private Map<String, String> properties;

@JsonAnyGetter(enabled = false)

public Map<String, String> getProperties() {

return properties;

}

}

Output:

{

"name":"My bean",

"properties":{"attr2":"val2","attr1":"val1"}

}

@JsonGetter

The @JsonGetter annotation is an alternative to the @JsonProperty annotation, which marks a method as a getter method.