## Reference links

<https://www.json.org/json-en.html>

<https://www.baeldung.com/jackson>

<https://github.com/eugenp/tutorials/tree/master/jackson-simple>

All annotations descriptions are here

<https://springframework.guru/jackson-annotations-json/>

<https://www.tutorialspoint.com/jackson_annotations/jackson_annotations_jsonpropertyorder.htm>

<https://www.baeldung.com/jackson-object-mapper-tutorial>

json to yaml

<https://www.bairesdev.com/tools/json2yaml/>

Dependencies

If jackson-databind anno is added then core and anno jars will come automatically as they are transitive

The original Jackson was made by Codehaus but is no longer supported.

The current version by FasterXML is the current version, and it supports XML, JSON, and YAML.

|  |  |
| --- | --- |
|  |  |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.16.0</version>  </dependency> | This is the most common and often the only dependency you need to include for general JSON processing.  It provides the ObjectMapper class, which is your main entry point for converting Java objects to JSON and vice-versa.  It automatically pulls in jackson-core and jackson-annotations as transitive dependencies. |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-core</artifactId>  <version>2.16.0</version> </dependency> | This provides the low-level streaming API (JSON parser and generator). jackson-databind depends on this |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-annotations</artifactId>  <version>2.16.0</version> </dependency> | This contains annotations like @JsonProperty, @JsonIgnore, etc., that you can use to customize how your Java objects are serialized/deserialized. jackson-databind also depends on this. |
| <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-yaml</artifactId>  <version>2.19.0</version>  </dependency> |  |
| <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-xml</artifactId>  <version>2.19.0</version>  </dependency> |  |
| <dependencyManagement>  <dependencies>  <dependency>  <groupId>com.fasterxml.jackson</groupId>  <artifactId>jackson-bom</artifactId>  <version>2.19.0</version> <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> | For larger projects or when you need multiple Jackson modules, it's highly recommended to use the Jackson Bill of Materials (BOM). This helps you manage compatible versions of all Jackson modules without explicitly specifying a version for each. |

## Json to java pojo

1. U should write a json schema file like below using this only we can generate the java pojos

{

"type": "object",

"javaType": "guru.learningjournal.kafka.examples.types.LineItem",

"properties": {

"ItemCode": {"type": "string"},

"ItemDescription": {"type": "string"},

"ItemPrice": {"type": "number"},

"ItemQty": {"type": "integer"},

"TotalValue": {"type": "number"}

}

}

1. U should add this plugin to generate the java classes from schema files using below maven code

<plugin>

<groupId>org.jsonschema2pojo</groupId>

<artifactId>jsonschema2pojo-maven-plugin</artifactId>

<version>0.5.1</version>

<executions>

<execution>

<goals>

<goal>generate</goal>

</goals>

<configuration>

<sourceDirectory>${project.basedir}/src/main/resources/schema/</sourceDirectory>

<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>

<includeAdditionalProperties>false</includeAdditionalProperties>

<includeHashcodeAndEquals>false</includeHashcodeAndEquals>

<generateBuilders>true</generateBuilders>

</configuration>

</execution>

</executions>

</plugin>

## Convert object to json string

**While converting make sure the target class should have getters and setters**

**ObjectMapper** mapper = **new** **ObjectMapper**(); **MyDto** dtoObject = **new** **MyDto**(); **String** dtoAsString = mapper.writeValueAsString(dtoObject);

Convert Json String to javaobject

|  |  |
| --- | --- |
| ObjectMapper mapper = new ObjectMapper();  String json = "{\"id\" : 1}";  Student student = mapper.readerFor(Student.class).readValue(json);  System.out.println(student.getTheId()); | // Parse a JSON string into a Java object  MyObject obj = mapper.readValue(jsonString, MyObject.class); |

## Convert obj to byte array

byte[] b= objectMapper.writeValueAsBytes(empObject);

if this is not there then 1st we will convert from object to string then string to bytes

= mapper.writeValueAsString(dtoObject).getBytes();

Convert to JSonNode

*JsonNode* jsonNode = objectMapper.readTree(objectMapper.writeValueAsBytes(s));

# Reading json data in 5 ways

## Sample json data

|  |  |
| --- | --- |
| {  "name": "John Smith",  "age": 30,  "city": "New York",  "isStudent": false,  "address": {  "street": "123 Main St",  "zipCode": "10001"  },  "hobbies": ["reading", "hiking", "coding"]    } |  |

Extracting values from a JSON object in Java can be achieved using various libraries and classes, including ObjectMapper, JSONPath, JSONObject, JsonReader, and JsonValue.

Using ObjectMapper –best way

The ObjectMapper class can be used to read JSON data into a Java object. For example, the readValue method can be used to parse a JSON string into a Java object.

This approach requires the JSON data to be in a specific format that matches the Java object's structure.

|  |  |
| --- | --- |
| Reading and binding to a class | Just reading to map / read as full tree and extract which ever values u want |
| // Import the ObjectMapper class  import com.fasterxml.jackson.databind.ObjectMapper;  // Create an instance of ObjectMapper  ObjectMapper mapper = new ObjectMapper();  // Parse a JSON string into a Java object  MyObject obj = mapper.readValue(jsonString, MyObject.class); | File file = Paths.*get*("src/main/resources/mani.JSON").toFile(); ObjectMapper mapper=new ObjectMapper();  *//Reading as full tree JsonNode* jsonNode = mapper.readTree(file); String text = jsonNode.path("address").path("zipCode").asText(); System.***out***.println(text); |

## Using JSONPath [2](https://learning.oreilly.com/library/view/json-at-work/9781491982389/ch04.html#source-card-2)

The JSONPath class can be used to extract specific values from a JSON object. For example, the getInt method can be used to extract an integer value from a JSON object.

// Import the JSONPath class

import io.restassured.path.json.JsonPath;

// Create an instance of JSONPath

JsonPath jsonPath = new JsonPath(jsonString);

// Extract an integer value from the JSON object

int value = jsonPath.getInt("path.to.value");

Code sample generated by AI. Please double-check responses.

This approach allows for flexible extraction of values from JSON data.

## Using JSONObject by google [3](https://learning.oreilly.com/library/view/json-at-work/9781491982389/ch04.html#source-card-3)

Always google will provide simple methods, to fetch like get methods

|  |  |
| --- | --- |
| <dependency>  <groupId>com.googlecode.json-simple</groupId>  <artifactId>json-simple</artifactId>  <version>1.1</version>  </dependency> |  |

The JSONObject class can be used to parse JSON data and extract values. For example, the get method can be used to extract a value from a JSON object.

|  |  |
| --- | --- |
| sample input  {"accountNumber": 100, "name": "Devi", "balance": 24.98}  String data="{\"accountNumber\": 100, \"name\": \"Devi\", \"balance\": 24.98}"; JSONObject obj = (JSONObject)JSONValue.*parse*(data); System.***out***.println(obj.get("name")); |  |

This approach requires the JSON data to be in a specific format that matches the JSONObject's structure.

## Using JsonReader [4](https://learning.oreilly.com/library/view/json-at-work/9781491982389/ch04.html#source-card-4)

The JsonReader class can be used to read JSON data from a string or file. For example, the readObject method can be used to parse a JSON string into a Java object.

// Import the JsonReader class

import javax.json.Json;

import javax.json.JsonReader;

// Create an instance of JsonReader

JsonReader jsonReader = Json.createReader(new StringReader(jsonString));

// Parse a JSON string into a Java object

JsonObject jsonObject = jsonReader.readObject();

This approach allows for flexible reading of JSON data from various sources.

## Using JsonValue [5](https://learning.oreilly.com/library/view/json-at-work/9781491982389/ch04.html#source-card-5)

The JsonValue class can be used to get the JSON value as a Java object. For example, the getValue method can be used to extract a value from a JSON object.

// Import the JsonValue class

import mjson.Json;

// Create an instance of JsonValue

Json json = Json.read(jsonString);

// Extract a value from the JSON object

Object value = json.getValue();

# **All Json annotations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| To ignore   * @JsonIgnore @JsonIgnoreProperties @JsonIgnoreType | * While serialization * @JsonValue * @JsonInclude * @JsonGetter * @JsonAnyGetter * @JsonPropertyOrder * @JsonRawValue * @JsonSerialize * @JsonRootName | Deserialization anno   * @JsonAnySetter * @JsonCreator * @JacksonInject * @JsonDeserialize | General annotations   * @JsonProperty * @JsonFormat * @JsonUnwrapped * @JsonView * @JsonManagedReference   @JsonBackReference  @JsonIdentityInfo   * @JsonFilter |  |

To print json correctly

ObjectMapper sm=new ObjectMapper();  
Student s=new Student();  
System.***out***.println(sm.writerWithDefaultPrettyPrinter().writeValueAsString(s));

{

"personId" : 0,

"name" : "James Clark",

"address" : {"doorNumber": 1234, "street": "phase-1", "city": "New York"}

}

Instead of printing data in single line it will print data in good format

### @JsonIgnore @JsonIgnoreProperties @JsonIgnoreType

This @JsonIgnore anno is to ignore that field during serialization and deserialization, so that while converting that obj to json this field will not be included

Among this @JsonIgnore is best, which can be written on a field

While deserializing also even if the file contains this data, that data will not be mapped or injected to java pojo’s or object

@JsonIgnoreProperties({"userId", "gender"}) – means ignore these properties during serialization and deserialization

while serializing these 2 fields will not be written to file &

While deserializing (reading from file and mapping to java classes) also even the json data contains these 2 fields, these will be ignored and these will not be injected into java object.

Whereas the problem with @JsonIgnoreProperties({"userId", "gender"}) is we have to manually write the fields, in future if we refactor field names in code it will be difficult to change here as well, these fields in anno will not be refactored automatically

|  |
| --- |
| import com.fasterxml.jackson.annotation.JsonIgnore;  @Builder public class Employee {  @JsonIgnore // now this field will not be serialized while converting to json  String **name**;  String **id**;  String **baseBranch**;   public static void main(String[] *args*) throws JsonProcessingException {  Employee emp= Employee.*builder*().name("Mani").id("2000").baseBranch("Hyd").build();  ObjectMapper mapper = new ObjectMapper();  mapper.setVisibility(*PropertyAccessor*.***ALL***, JsonAutoDetect.*Visibility*.***ANY***);  System.***out***.println(mapper.writeValueAsString(emp));  } |

@JsonAutoDetect

This anno is to detect all these variables (by default only public variables or public getter methods fields only will be detected & serialized – either variable or method must be public)

If that variable is private u must tell specifically to serialize the private variables by specifying the auto detect

U need this for Jackson latest versions, now by default private variables will not be serialized (if public getter is absent),

The @JsonAutoDetect annotation is used at the class level to tell Jackson to override the visibility of the properties of a class during serialization and deserialization.

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***) – means even the private variables will also be detected and serialized (if that var is already having public getter no need of this explicit detection)

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.NON\_PRIVATE)  
public class Employee {  
}

Now if class have atleast 1 public variable or getter or setter methods, it wont throw any exception, but if all non public are there means we should add this anno

You can set the visibility with the following elements:

creatorVisibility

fieldVisibility

getterVisibility

setterVisibility

isGetterVisibility

The JsonAutoDetect class defines public static constants that are similar to Java class visibility levels. They are:

ANY

DEFAULT

NON\_PRIVATE

NONE

PROTECTED\_AND\_PRIVATE

PUBLIC\_ONLY

Let us consider an example of Java class that uses the @JsonAutoDetect annotation.

|  |  |
| --- | --- |
| import com.fasterxml.jackson.annotation.JsonAutoDetect;  @JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)  //Bec of this all the fields even private fields also will be serialized  public class AutoDetectDemoBean {  private long personId = 123L;  private String name = "James Clark";  public long getPersonId() {  return personId;  }  public void setPersonId(long personId) {  this.personId = personId;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "IgnoreDemoBean{" +  "personId=" + personId +  ", name='" + name + '\'' +  '}';  }  }  Output  Now both person and name will be serialized | @Builder @JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.NON\_PRIVATE)  //means all fields with non-private only will be serialized public class Employee {  @JsonIgnore  String **name**;  private String **id**;  public String **baseBranch**;   public static void main(String[] *args*) throws JsonProcessingException {  Employee emp = Employee.*builder*().name("Mani").id("2000").baseBranch("Hyd").build();  ObjectMapper mapper = new ObjectMapper();  System.***out***.println(mapper.writeValueAsString(emp));  } }  Output  {"baseBranch":"Hyd"}  Here out of 3 fields, 1 is already ignored another id field is private, but we selected only non private, hence this field also will be ignored, only base branch will be serialized |
| *@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)*  public class Student {   public long **personId**=0;  **private** String **name** ="Manideep" ;  } | {  "personId" : 0,  "name" : "Manideep"  }  // as we added that autoDetect anno it detected private var and serialized, else it wouldn’t have been serialized |

Serialization annotations

@JsonValue –not used

Keep this anno on method so that that method will be called to serialize

Never used - The @JsonValue annotation is used at the method level. This annotation tells Jackson to use this method to generate the JSON string from the Java object.

Actually we can generate json using “new objectMapper.writeValueAsString(object data)”, but if u annotate this on method that method will be executed to generate json

@JsonValue

public String toJson(){

return this.name+","+this.personId+","+this.toString();

}

@Override

public String toString() {

return "ValueDemoBean{" +

"personId=" + personId +

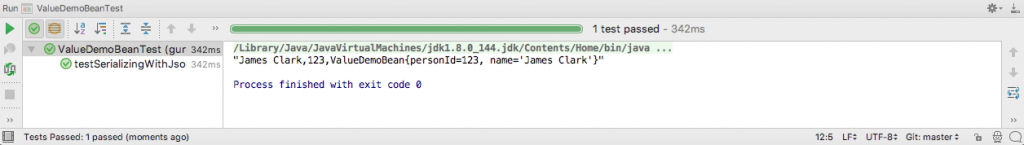
", name='" + name + '\'' +

'}';

}

Response:- if u observe here our method is called while serializing

String jsonString = objectMapper.writeValueAsString(new ValueDemoBean());



@JsonInclude(Include.NON\_NULL)

By default all nulls (if value is null for that key), non null value entries will be serialized means all key with value either null or empty also ok, it will be serialized when converting to json, if u want to customize that behaviour then use this anno

@JsonInclude(JsonInclude.*Include*.***NON\_NULL***) – means don’t include null

@JsonInclude(JsonInclude.*Include*.***NON\_EMPTY***) – means don’t include empty values

If u want to stop that behaviour then customize

[NON\_NULL](https://fasterxml.github.io/jackson-annotations/javadoc/2.6/com/fasterxml/jackson/annotation/JsonInclude.Include.html#NON_NULL)

Value that indicates that only properties with non-null values are to be included means null values will be ignored

The @JsonInclude annotation is used to exclude properties or fields of a class under certain conditions. This is defined using the JsonInclude.Include enum. This enum contains constants, that determine whether or not to exclude the property. The constants are:

ALWAYS, NON\_DEFAULT, NON\_EMPTY (don’t include empty values also), NON\_NULL (include non null means – don’t include null at all)

@JsonInclude(JsonInclude.Include.NON\_NULL)

public class Foo {

@JsonInclude(JsonInclude.Include.NON\_NULL)

private String fieldOne;

@JsonInclude(JsonInclude.Include.ALWAYS)

private String fieldTwo;

}

ObjectMapper mapper = new ObjectMapper();

mapper.configure(SerializationFeature.WRITE\_NULL\_MAP\_VALUES, false);

**ObjectMapper** mapper = **new** **ObjectMapper**(); mapper.setSerializationInclusion(Include.NON\_NULL);

Ex:-

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***)  
@JsonInclude(JsonInclude.*Include*.***NON\_NULL***) //We said don’t include null – means only nulls will be excluded, empty will be included happily  
public class Employee {  
*// @JsonIgnore* String **name**;  
 public String **id**;  
 public String **baseBranch**;  
  
 public static void main(String[] *args*) throws JsonProcessingException {  
 Employee emp = Employee.*builder*().id("").name("Mani").baseBranch("Hyd").build();  
 ObjectMapper mapper = new ObjectMapper();  
 System.***out***.println(mapper.writeValueAsString(emp));  
 }  
}

@JsonProperty("employeeName")

This is to rename the field while serializing (serialize with diff name)and helps while deserializing (if json string have diff prop and if u want to map to var with diff name)

The @JsonProperty annotation is used to map property names with JSON keys during serialization and deserialization. By default, if you try to serialize a POJO, the generated JSON will have keys mapped to the fields of the POJO. If you want to override this behavior, you can use the @JsonProperty annotation on the fields. It takes a String attribute that specifies the name that should be mapped to the field during serialization.

You can also use @JsonProperty annotation during deserialization when the property names of the JSON and the field names of the Java object do not match.This is to rename the field while serializing

Example

@JsonInclude(JsonInclude.Include.*NON\_NULL*)  
class Employee {  
 @JsonProperty("employeeName") //Due to this annotation while converting from object to json the fields will be renamed to employeeName instead of variable name  
 String empname;  
 Integer empid;  
 String address;

}

ObjectMapper mapper = new ObjectMapper();  
Employee emp=new Employee("manideep",12,null);//Here we are passing null for address, as we ignored null fields this null value won’t appear in console  
String json = mapper.writeValueAsString(emp);  
System.out.println(json);

Output :- here it is printed as employeeName instead of variable name (empName)

{"empid":12,"employeeName":"manideep"}

1. @JsonProperty(“customName other than variable name”)

|  |  |
| --- | --- |
| //While serializing  public class College {   @JsonProperty("collegeName")  String **colName**="St.Anns";  @JsonProperty("collegeLocation")  String **colLocation** ="Chirala";   public static void main(String[] *args*) throws Exception {  ObjectMapper objectMapper= new ObjectMapper();  College sd=new College();  String objAsJson = objectMapper.writeValueAsString(sd);  System.***out***.println(objAsJson);  College o = (College)objectMapper.readerFor(College.class).readValue(objAsJson);  System.***out***.println("parsed from string to obj \n"+o);  } | {"collegeName":"St.Anns","collegeLocation":"Chirala"}  parsed from string to obj  College{colName='St.Anns', colLocation='Chirala'} |
| While Deserialing  @JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***) public class College {    @JsonProperty("collegeName")  String **colName**="St.Anns";  @JsonProperty("collegeLocation")  String **colLocation** ="Chirala";   public static void main(String[] *args*) throws Exception {  ObjectMapper objectMapper= new ObjectMapper(); **//Here deserialized string has a diff name and java variables have diff name, still we can map with @JsonProperty anno**  String objAsJson= "{\"collegeName\":\"St.Anns\",\"collegeLocation\":\"Chirala\"}";  College o = (College)objectMapper.readerFor(College.class).readValue(objAsJson);  System.***out***.println("parsed from string to obj \n"+o);  } |  |

#### @JsonGetter

The @JsonGetter annotation is used to customize the generated JSON keys (same like @JsonProperty()). But this anno should be placed on getter method

This is accomplished with the value argument of @JsonGetter. The value passed is the name that should be used as the JSON key.

//package guru.springframework.blog.jsonannotation.domain.serialization;

import com.fasterxml.jackson.annotation.JsonGetter;

public class GetterDemoBean {

public long personId = 123L;

public String personName = "James Clark";

@JsonGetter(value = "person-id")

public long getPersonId() {

return personId;

}

@JsonGetter(value = "person-name")

public String getPersonName() {

return personName;

}

}



@JsonAnyGetter - for map var

This is useful when u are using map variable, instead of declaring all single variable we can fill map and

Note:- this is not working for me as if I keep this anno its creating duplicate elements,

if I just declare the map and serialize, its automatically perfectly serializing without creating any duplicate elements

The @JsonAnyGetter annotation can be used when you don’t want to declare a property or a method for every possible key in JSON. This annotation is used on the getter methods, which enables you to use a Map to hold all your properties that you want to serialize.

|  |  |
| --- | --- |
| Here I added annotation;- in result its creating me duplicate elements  public class JsonAnyGetterDemo {  public *Map*<String,String> **hm**=*Map*.*of*("city","kavali","district","Nellore");   @JsonAnyGetter  public *Map*<String, String> getHm() {  return **hm**;  }   public static void main(String[] *args*) throws Exception{  ObjectMapper objectMapper=new ObjectMapper();  System.***out***.println(objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(new JsonAnyGetterDemo()));  } }  result;- here district and city elements are duplicated  {  "hm" : {  "district" : "Nellore",  "city" : "kavali"  },  "district" : "Nellore",  "city" : "kavali"  } | Deleted this annotation  public class JsonAnyGetterDemo {  public *Map*<String,String> **hm**=*Map*.*of*("city","kavali","district","Nellore");   public static void main(String[] *args*) throws Exception{  ObjectMapper objectMapper=new ObjectMapper();  System.***out***.println(objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(new JsonAnyGetterDemo()));  } }  result: - perfectly works fine without duplication  {  "hm" : {  "district" : "Nellore",  "city" : "kavali"  }  }  For Deserializing also  public class JsonAnyGetterDemo {  public *Map*<String,String> **hm**;   public static void main(String[] *args*) throws Exception{  ObjectMapper objectMapper=new ObjectMapper();  JsonAnyGetterDemo o = objectMapper.readerFor(JsonAnyGetterDemo.class).readValue("{\"hm\":{\"city\":\"kavali\",\"district\":\"Nellore\"}}\n");  System.***out***.println(o.**hm**);  } } |

@JsonPropertyOrder

The @JsonPropertyOrder annotation tells Jackson to serialize the Java object to JSON, in the order specified as the arguments of the annotation. This annotation also allows partial ordering. The properties are first serialized in the order, in which they are found. Followed by any other properties not included in the annotation.

import com.fasterxml.jackson.annotation.JsonPropertyOrder;

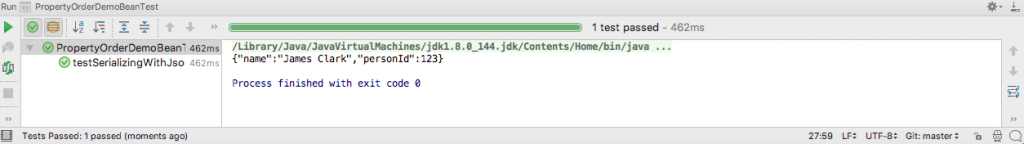
@JsonPropertyOrder({"name", "personId"})

public class PropertyOrderDemoBean {

public long personId = 123L;

public String name = "James Clark";

}



@JsonRawValue

Means serialize as it is , don’t include double quotes

|  |  |
| --- | --- |
| @JsonInclude(JsonInclude.*Include*.***NON\_EMPTY***) public class Student {   public long **personId**=0;  public String **name** ;  @JsonRawValue  public String **address** = "{\"doorNumber\": 1234, \"street\": \"phase-1\", " +  "\"city\": \"New York\"}";  public static void main(String[] *args*) throws Exception {  ObjectMapper sm=new ObjectMapper();  Student s=new Student();  System.***out***.println(sm.writerWithDefaultPrettyPrinter().writeValueAsString(s));  } } | If anno is missing – |

|  |  |
| --- | --- |
| Without @JsonRawValue - by default all keys and values will be enclosed in “ ” double quotes | {  "personId" : 0,  "address" : "{\"doorNumber\": 1234, \"street\": \"phase-1\", \"city\": \"New York\"}"  } |
| If anno is present - raw will be printed as it is without double quotes | {  "personId" : 0,  "address" : {"doorNumber": 1234, "street": "phase-1", "city": "New York"}  } |

@JsonSerialize

This is to specify custom serializer to serialize custom variables

If class is having normal variables like string, primitives, class types (like employee class having address) it will easily serialize,

if it contains LocalDateTime we should provide custom serializer to serialize that fields

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***)  
public class SerializeDemo {  
 **LocalDateTime ldt= LocalDateTime.*now*(); -- By default as this var cannot be serialized unless u use a custom serializer**  
 private String **cityName**="hyd";  
  
 public static void main(String[] *args*) throws Exception {  
 ObjectMapper objectMapper= new ObjectMapper();  
 SerializeDemo sd=new SerializeDemo();  
 System.***out***.println(objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(sd));  
 }  
}

Output: exception as default serializer cant serialize

Sample code

|  |  |
| --- | --- |
| @JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***) public class SerializeDemo {  @JsonSerialize(using = MyLDTSerializer.class)  LocalDateTime **ldt**= LocalDateTime.*now*();  private String **cityName**="hyd";   public static void main(String[] *args*) throws Exception {  ObjectMapper objectMapper= new ObjectMapper();  SerializeDemo sd=new SerializeDemo();  System.***out***.println(objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(sd));  } }  Output  {  "ldt" : "2024-05-05T07:24:22.308317500",  "cityName" : "hyd"  } | public class MyLDTSerializer extends *StdSerializer*<LocalDateTime> {  protected MyLDTSerializer(Class<LocalDateTime> *t*) {  super(*t*);  }  protected MyLDTSerializer() {  this(null);  }  @Override  public void serialize(LocalDateTime *localDateTime*, *JsonGenerator jsonGenerator*, *SerializerProvider serializerProvider*) throws IOException {  *jsonGenerator*.writeString(*localDateTime*.toString());  } } |

@JsonRootName

This anno is to enclose root object with given key

@JsonRootName("personDetails")

objectMapper.enable(SerializationFeature.WRAP\_ROOT\_VALUE)

The @JsonRootName annotation can be used to tell Jackson to wrap the object to be serialized with a top-level element. You can pass the name as a parameter to the @JsonRootName annotation.

|  |  |
| --- | --- |
| @JsonAutoDetect(fieldVisibility = JsonAutoDetect.*Visibility*.***ANY***) @JsonRootName("personDetails") public class SerializeDemo {  @JsonSerialize(using = MyLDTSerializer.class)  @JsonProperty(value = "dateOfBirth")  LocalDateTime **ldt**= LocalDateTime.*now*();  private String **cityName**="hyd";   public static void main(String[] *args*) throws Exception {  ObjectMapper objectMapper= new ObjectMapper();  objectMapper.enable(*SerializationFeature*.***WRAP\_ROOT\_VALUE***);  SerializeDemo sd=new SerializeDemo();  System.***out***.println(objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(sd));  } } | response  {  "personDetails" : {  "cityName" : "hyd",  "dateOfBirth" : "08:07:07.7342997"  }  } |

Deserialization Annotations

# Yaml

Why yaml is preferred, because in yaml

1. no need of double quotes, commas , square braces….

|  |  |
| --- | --- |
| {  "json": [  "rigid",  "better for data interchange"  ],  "yaml": [  "slim and flexible",  "better for configuration"  ],  "object": {  "key": "value",  "array": [  {  "null\_value": null  },  {  "boolean": true  },  {  "integer": 1  },  {  "alias": "aliases are like variables"  },  {  "alias": "aliases are dflike variables"  }  ]  },  "paragraph": "Blank lines denote\nparagraph breaks\n",  "content": "Or we\ncan auto\nconvert line breaks\nto save space",  "alias": {  "bar": "baz"  },  "alias\_reuse": {  "bar": "baz"  }  } | ---  json:  - rigid  - better for data interchange  yaml:  - slim and flexible  - better for configuration  object:  key: value  array:  - null\_value:  - boolean: true  - integer: 1  - alias: aliases are like variables  - alias: aliases are dflike variables  paragraph: |  Blank lines denote  paragraph breaks  content: |-  Or we  can auto  convert line breaks  to save space  alias:  bar: baz  alias\_reuse:  bar: baz |

Yaml is the super set of json means yaml have more properties that json ex:- in yaml we can comment using # but it’s not possible in json

In yaml we can write double quotes in value, it’s ok

### Yaml rules

https://yaml.org/

1. Value must start with space as below example

name: Manideep

1. If that property belongs to parent , then child should start with 2 space, EVEN for list also child elements should start 2 spaces or tab

Array ante add geetha in yaml

|  |  |
| --- | --- |
| {  "name": "eazy",  "person": {  "name": "manideep",  "age": 20,  "address": {  "doorno": "flat 302",  "area": "shaikpet",  "zip": 500008  },  "courses": [  "java",  "python",  "kubernetes"  ]  }  } | name: eazy  person:  name: manideep  age: 20  address:  doorno: flat 302  area: shaikpet  zip: 500008  courses:  - java  - python  - kubernetes |