Do JSON with Jackson

By Baeldung

TABLE OF CONTENTS

1: A GUIDE TO JACKSON ANNOTATIONS

2: JACKSON IGNORE PROPERTIES ON MARSHALLING

3: IGNORE NULL FIELDS WITH JACKSON

4: JACKSON – CHANGE NAME OF FIELD

5: JACKSON – MARSHALL STRING TO JSONNODE

6: JACKSON UNMARSHALLING JSON WITH UNKNOWN PROPERTIES

CHAPTER 1

Do JSON with Jackson

Introduction

This short and practical eBook is focused on the basics of Jackson and on getting JSON serialized and deserialized efficiently and with flexibility.

1: A GUIDE TO JACKSON ANNOTATIONS

1. Overview

In this section we'll do a deep dive into Jackson Annotations.

We'll see how to use the existing annotations, how to create custom ones and finally – how to disable them.

2. Jackson Serialization Annotations

First – let's take a look at the serialization annotations.

2.1. @JsonAnyGetter

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

Here's a quick example – the *ExtendableBean* entity has the *name* property and a set of extendable attributes in form of key/value pairs:

```
public class ExtendableBean {
   public String name;
   private Map<String, String> properties;

   @JsonAnyGetter
   public Map<String, String> getProperties() {
      return properties;
   }
}
```

When we serialize an instance of this entity, we get all the key-values in the *Map* as standard, plain properties:

```
{
   "name":"My bean",
   "attr2":"val2",
   "attr1":"val1"
}
```

And here how the serialization of this entity looks like in practice:

```
@Test
public void whenSerializingUsingJsonAnyGetter_thenCorrect()
  throws JsonProcessingException {
   ExtendableBean bean = new ExtendableBean("My bean");
   bean.add("attr1", "val1");
   bean.add("attr2", "val2");

   String result = new ObjectMapper().writeValueAsString(bean);
   assertThat(result, containsString("attr1"));
   assertThat(result, containsString("val1"));
}
```

2.2. @JsonGetter

The *@JsonGetter* annotation is an alternative to *@JsonProperty* annotation to mark the specified method as a getter method.

In the following example – we specify the method *getTheName()* as the getter method of *name* property of *MyBean* entity:

```
public class MyBean {
   public int id;
   private String name;

   @JsonGetter("name")
   public String getTheName() {
      return name;
   }
}
```

And here's how this works in practice:

```
@Test
public void whenSerializingUsingJsonGetter_thenCorrect()
  throws JsonProcessingException {
   MyBean bean = new MyBean(1, "My bean");

   String result = new ObjectMapper().writeValueAsString(bean);
   assertThat(result, containsString("My bean"));
   assertThat(result, containsString("1"));
}
```

2.3. @JsonPropertyOrder

The *@JsonPropertyOrder* annotation is used to specify the order of properties on serialization.

Let's set a custom order for the properties of a *MyBean* entity:

```
@JsonPropertyOrder({ "name", "id" })
public class MyBean {
    public int id;
    public String name;
}
```

And here is the output of serialization:

```
{
    "name":"My bean",
    "id":1
}
```

And a simple test:

```
@Test
public void whenSerializingUsingJsonPropertyOrder_thenCorrect()
  throws JsonProcessingException {
    MyBean bean = new MyBean(1, "My bean");

    String result = new ObjectMapper().writeValueAsString(bean);
    assertThat(result, containsString("My bean"));
    assertThat(result, containsString("1"));
}
```

2.4. @JsonRawValue

@JsonRawValue is used to instruct the Jackson to serialize a property exactly as is.

In the following example – we use *@JsonRawValue* to embed some custom JSON as value of an entity:

```
public class RawBean {
   public String name;

   @JsonRawValue
   public String json;
}
```

The output of serializing the entity is:

```
{
   "name":"My bean",
   "json":{
      "attr":false
   }
}
```

And a simple test:

```
@Test
public void whenSerializingUsingJsonRawValue_thenCorrect()
  throws JsonProcessingException {
   RawBean bean = new RawBean("My bean", "{"attr":false}");

   String result = new ObjectMapper().writeValueAsString(bean);
   assertThat(result, containsString("My bean"));
   assertThat(result, containsString("{"attr":false}"));
}
```

2.5. @JsonValue

@JsonValue indicates a single method that should be used to serialize the entire instance.

For example in an **enum** – we annotate the *getName* with *@JsonValue* so that any such entity is serialized via its name:

```
public enum TypeEnumWithValue {
   TYPE1(1, “Type A”), TYPE2(2, “Type 2”);

   private Integer id;
   private String name;

   @JsonValue
   public String getName() {
      return name;
   }
}
```

Our test:

```
@Test
public void whenSerializingUsingJsonValue_thenCorrect()
  throws JsonParseException, IOException {
   String enumAsString =
     new ObjectMapper().writeValueAsString(TypeEnumWithValue.TYPE1);

   assertThat(enumAsString, is(““Type A””));
}
```

2.6. @JsonRootName

The *@JsonRootName* annotation is used – if wrapping is enabled – to specify the name of the root wrapper to be used.

Wrapping means that instead of serializing a *User* to something like:

```
{
   "id": 1,
   "name": "John"
}
```

It's going to be wrapped like this:

```
{
   "User": {
      "id": 1,
      "name": "John"
   }
}
```

So, let's look at an example – we're going to use the *@JsonRootName* annotation to indicate the name of this potential wrapper entity:

```
@JsonRootName(value = "user")
public class UserWithRoot {
   public int id;
   public String name;
}
```

By default, the name of the wrapper would be the name of the **class** – *UserWithRoot*. By using the annotation, we get the cleaner looking user:

```
@Test
public void whenSerializingUsingJsonRootName_thenCorrect()
  throws JsonProcessingException {
   UserWithRoot user = new User(1, "John");

   ObjectMapper mapper = new ObjectMapper();
   mapper.enable(SerializationFeature.WRAP_ROOT_VALUE);
   String result = mapper.writeValueAsString(user);

   assertThat(result, containsString("John"));
   assertThat(result, containsString("user"));
}
```

Here is the output of serialization:

```
{
   "user":{
      "id":1,
      "name":"John"
   }
}
```

2.7. @JsonSerialize

@JsonSerialize is used to indicate a custom serializer will be used to marshall the entity.

Let's look at a quick example – we're going to use *@JsonSerialize* to serialize the eventDate property with a *CustomDateSerializer*:

```
public class Event {
   public String name;

   @JsonSerialize(using = CustomDateSerializer.class)
   public Date eventDate;
}
```

Here's the simple custom Jackson serializer:

```
public class CustomDateSerializer extends JsonSerializer<Date> {

   private static SimpleDateFormat formatter =
     new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");

   @Override
   public void serialize(Date value, JsonGenerator gen, SerializerProvider arg2)
     throws IOException, JsonProcessingException {
       gen.writeString(formatter.format(value));
   }
}
```

Let's use these in a test:

```
@Test
public void whenSerializingUsingJsonSerialize_thenCorrect()
  throws JsonProcessingException, ParseException {
    SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");

    String toParse = "20-12-2014 02:30:00";
    Date date = df.parse(toParse);
    Event event = new Event("party", date);

    String result = new ObjectMapper().writeValueAsString(event);
    assertThat(result, containsString(toParse));
}
```

3. Jackson Deserialization Annotations

Next – let's explore the Jackson deserialization annotations.

3.1. @JsonCreator

The *@JsonCreator* annotation is used to tune the constructor/factory used in deserialization.

It's very helpful when we need to deserialize some JSON that doesn't exactly match the target entity we need to get.

Let's look at an example; say we need to deserialize the following JSON:

```
{
    "id":1,
    "theName":"My bean"
}
```

However, *there is no theName field in our target entity* – there is only a name field. Now – we don't want to change the entity itself – we just need a little more control over the unmarshalling process – by annotating the constructor with *@JsonCreator* and using the *@JsonProperty* annotation as well:

```
public class BeanWithCreator {
    public int id;
    public String name;

    @JsonCreator
    public BeanWithCreator(
      @JsonProperty("id") int id,
      @JsonProperty("theName") String name) {
        this.id = id;
        this.name = name;
    }
}
```

Let's see this in action:

```
@Test
public void whenDeserializingUsingJsonCreator_thenCorrect()
  throws JsonProcessingException, IOException {
    String json = "{"id":1,"theName":"My bean"}";

    BeanWithCreator bean =
      new ObjectMapper().reader(BeanWithCreator.class).readValue(json);
    assertEquals("My bean", bean.name);
}
```

3.2. @JacksonInject

@JacksonInject is used to indicate a property that will get its value from injection and *not from the JSON data*.

In the following example – we use *@JacksonInject* to inject the property *id*:

```
public class BeanWithInject {
    @JacksonInject
    public int id;

    public String name;
}
```

And here's how this works:

```
@Test
public void whenDeserializingUsingJsonInject_thenCorrect()
  throws JsonProcessingException, IOException {
    String json = "{"name":"My bean"}";

    InjectableValues inject = new InjectableValues.Std().addValue(int.class, 1);
    BeanWithInject bean = new ObjectMapper().reader(inject)
                        .forType(BeanWithInject.class)
                        .readValue(json);

    assertEquals("My bean", bean.name);
    assertEquals(1, bean.id);
}
```

3.3. @JsonAnySetter

@JsonAnySetter allows you the flexibility of using a Map as standard properties. On de-serialization, the properties from JSON will simply be added to the map.

Let's see how this works – we'll use *@JsonAnySetter* to deserialize the entity *ExtendableBean*:

```
public class ExtendableBean {
   public String name;
   private Map<String, String> properties;

   @JsonAnySetter
   public void add(String key, String value) {
      properties.put(key, value);
   }
}
```

This is the JSON we need to deserialize:

```
{
   "name":"My bean",
   "attr2":"val2",
   "attr1":"val1"
}
```

And here's how this all ties in together:

```
@Test
public void whenDeserializingUsingJsonAnySetter_thenCorrect()
  throws JsonProcessingException, IOException {
   String json = "{"name":"My bean","attr2":"val2","attr1":"val1"}";

   ExtendableBean bean =
     new ObjectMapper().reader(ExtendableBean.class).readValue(json);

   assertEquals("My bean", bean.name);
   assertEquals("val2", bean.getProperties().get("attr2"));
}
```

3.4. @JsonSetter

@JsonSetter is an alternative to *@JsonProperty* – used to mark the a method as a setter method.

This is super useful when we need to read some JSON data but *the target entity* **class** *doesn't exactly match that data* and so we need to tune the process to make it fit.

In the following example, we'll specify the method *setTheName()* as the setter of the name property in our *MyBean* entity:

```
public class MyBean {
   public int id;
   private String name;

   @JsonSetter("name")
   public void setTheName(String name) {
      this.name = name;
   }
}
```

Now, when we need to unmarshall some JSON data – this works perfectly well:

```
@Test
public void whenDeserializingUsingJsonSetter_thenCorrect()
  throws JsonProcessingException, IOException {
   String json = "{"id":1,"name":"My bean"}";

   MyBean bean =
     new ObjectMapper().reader(MyBean.class).readValue(json);
   assertEquals("My bean", bean.getTheName());
}
```

3.5. @JsonDeserialize

@JsonDeserialize is used to indicate *the use of a custom deserializer*.

Let's see how that plays out – we'll use *@JsonDeserialize* to deserialize the *eventDate* property with the *CustomDateDeserializer*:

```
public class Event {
    public String name;

    @JsonDeserialize(using = CustomDateDeserializer.class)
    public Date eventDate;
}
```

Here's the custom deserializer:

```
public class CustomDateDeserializer extends JsonDeserializer<Date> {

    private static SimpleDateFormat formatter =
      new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");

    @Override
    public Date deserialize(JsonParser jsonparser, DeserializationContext context)
      throws IOException, JsonProcessingException {
        String date = jsonparser.getText();
        try {
            return formatter.parse(date);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
}
```

And here's the back-to-back test:

```
@Test
public void whenDeserializingUsingJsonDeserialize_thenCorrect()
  throws JsonProcessingException, IOException {
    String json = "{"name":"party","eventDate":"20-12-2014 02:30:00"}";

    SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
    Event event = new ObjectMapper().reader(Event.class).readValue(json);

    assertEquals("20-12-2014 02:30:00", df.format(event.eventDate));
}
```

4. Jackson Property Inclusion Annotations

4.1. @JsonIgnoreProperties

@JsonIgnoreProperties – one of the most common annotations in Jackson – is used to mark a property or a list of properties *to be ignored at the* **class** *level.*

Let's go over a quick example ignoring the property *id* from serialization:

```
@JsonIgnoreProperties({ "id" })
public class BeanWithIgnore {
    public int id;
    public String name;
}
```

And here's the test making sure the ignore happens:

```
@Test
public void whenSerializingUsingJsonIgnoreProperties_thenCorrect()
  throws JsonProcessingException, IOException {
    BeanWithIgnore bean = new BeanWithIgnore(1, "My bean");

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

    assertThat(result, containsString("My bean"));
    assertThat(result, not(containsString("id")));
}
```

4.2. @JsonIgnore

The *@JsonIgnore* annotation is used to mark a property to be ignored at the field level.

Let's use *@JsonIgnore* to ignore the property *id* from serialization:

```
public class BeanWithIgnore {
    @JsonIgnore
    public int id;

    public String name;
}
```

And the test making sure that *id* was successfully ignored:

```
@Test
public void whenSerializingUsingJsonIgnore_thenCorrect()
  throws JsonProcessingException, IOException {
    BeanWithIgnore bean = new BeanWithIgnore(1, "My bean");

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

    assertThat(result, containsString("My bean"));
    assertThat(result, not(containsString("id")));
}
```

4.3. @JsonIgnoreType

@JsonIgnoreType is used to *mark all properties of annotated type to be ignored.*

Let's use the annotation to mark all properties of type *Name* to be ignored:

```
public class User {
   public int id;
   public Name name;

   @JsonIgnoreType
   public static class Name {
      public String firstName;
      public String lastName;
   }
}
```

Here's the simple test making sure the ignore works correctly:

```
@Test
public void whenSerializingUsingJsonIgnoreType_thenCorrect()
  throws JsonProcessingException, ParseException {
   User.Name name = new User.Name("John", "Doe");
   User user = new User(1, name);

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

   assertThat(result, containsString("1"));
   assertThat(result, not(containsString("name")));
   assertThat(result, not(containsString("John")));
}
```

4.4. @JsonInclude

@JsonInclude is used to *exclude properties with empty/null/default values*.

Let's look at an example – excluding nulls from serialization:

```
@JsonInclude(Include.NON_NULL)
public class MyBean {
   public int id;
   public String name;
}
```

Here's the full test:

```
public void whenSerializingUsingJsonInclude_thenCorrect()
  throws JsonProcessingException, IOException {
   MyBean bean = new MyBean(1, null);

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

   assertThat(result, containsString("1"));
   assertThat(result, not(containsString("name")));
}
```

4.5. @JsonAutoDetect

@JsonAutoDetect is used to override the default semantics of *which properties are visible and which are not.*

Let's take a look at how the annotation can be very helpful with a simple example – let's enable serializing private properties:

```
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
public class PrivateBean {
   private int id;
   private String name;
}
```

And the test:

```
@Test
public void whenSerializingUsingJsonAutoDetect_thenCorrect()
  throws JsonProcessingException, IOException {
    PrivateBean bean = new PrivateBean(1, “My bean”);

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

    assertThat(result, containsString(“1”));
    assertThat(result, containsString(“My bean”));
}
```

5. Jackson Polymorphic Type Handling Annotations

Next – let’s take a look at Jackson polymorphic type handling annotations:

@JsonTypeInfo is used to indicate details of what type information is included in serialization
@JsonSubTypes is used to indicate sub-types of annotated type
@JsonTypeName is used to define logical type name to use for annotated **class**

Let’s look over a more complex example and use all three – *@JsonTypeInfo, @JsonSubTypes and @JsonTypeName* – to serialize/desrialize the entity *Zoo*:

```
public class Zoo {
    public Animal animal;

    @JsonTypeInfo(use = JsonTypeInfo.Id.NAME,
                  include = As.PROPERTY,
                  property = “type”)
    @JsonSubTypes({
        @JsonSubTypes.Type(value = Dog.class, name = “dog”),
        @JsonSubTypes.Type(value = Cat.class, name = “cat”)
    })
    public static class Animal {
        public String name;
    }
```

```
    @JsonTypeName(“dog”)
    public static class Dog extends Animal {
        public double barkVolume;
    }

    @JsonTypeName(“cat”)
    public static class Cat extends Animal {
        boolean likesCream;
        public int lives;
    }
}
```

When we do serialization:

```
@Test
public void whenSerializingPolymorphic_thenCorrect()
  throws JsonProcessingException, IOException {
    Zoo.Dog dog = new Zoo.Dog(“lacy”);
    Zoo zoo = new Zoo(dog);

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

    assertThat(result, containsString(“type”));
    assertThat(result, containsString(“dog”));
}
```

Here’s what serializing the *Zoo* instance with the *Dog* will result in:

```
{
    “animal”: {
        “type”: “dog”,
        “name”: “lacy”,
        “barkVolume”: 0
    }
}
```

Now for de-serialization – let's start from the following JSON input:

```
   "animal":{
      "name":"lacy",
      "type":"cat"
   }
}
```

And let's see how that gets unmarshalled to a *Zoo* instance:

```
@Test
public void whenDeserializingPolymorphic_thenCorrect()
throws JsonProcessingException, IOException {
   String json = "{"animal":{"name":"lacy","type":"cat"}}";

   Zoo zoo = new ObjectMapper().reader()
                        .withType(Zoo.class)
                        .readValue(json);

   assertEquals("lacy", zoo.animal.name);
   assertEquals(Zoo.Cat.class, zoo.animal.getClass());
}
```

6. Jackson General Annotations

Next – let's discuss some of Jackson more general annotations.

6.1. @JsonProperty

@JsonProperty is used to indicate *the property name in JSON.*

Let's go over the annotation with a simple example – and use *@JsonProperty* to serialize/

desrialize 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;
   }
}
```

Our test:

```
@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().reader(MyBean.class)
                                .readValue(result);
   assertEquals("My bean", resultBean.getTheName());
}
```

6.2. @JsonFormat

The *@JsonFormat* annotation can be used to *specify a format when serializing Date/Time values.*

In the following example – we use *@JsonFormat* to control the format of the property *eventDate*:

```
public class Event {
   public String name;

   @JsonFormat(
     shape = JsonFormat.Shape.STRING,
     pattern = "dd-MM-yyyy hh:mm:ss")
   public Date eventDate;
}
```

And here's the test:

```
@Test
public void whenSerializingUsingJsonFormat_thenCorrect()
  throws JsonProcessingException, ParseException {
   SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
   df.setTimeZone(TimeZone.getTimeZone("UTC"));

   String toParse = "20-12-2014 02:30:00";
   Date date = df.parse(toParse);
   Event event = new Event("party", date);

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

   assertThat(result, containsString(toParse));
}
```

6.3. @JsonUnwrapped

@JsonUnwrapped is used to define that *a value should be unwrapped / flattened when serialized.*

Let's see exactly how that works; we'll use the annotation to unwrap the property *name*:

```
public class UnwrappedUser {
   public int id;

   @JsonUnwrapped
   public Name name;

   public static class Name {
      public String firstName;
      public String lastName;
   }
}
```

Let's now serialize an instance of this **class**:

```
@Test
public void whenSerializingUsingJsonUnwrapped_thenCorrect()
  throws JsonProcessingException, ParseException {
   UnwrappedUser.Name name = new UnwrappedUser.Name("John", "Doe");
   UnwrappedUser user = new UnwrappedUser(1, name);

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

   assertThat(result, containsString("John"));
   assertThat(result, not(containsString("name")));
}
```

Here's how the output looks like – the fields of the **static** nested **class** unwrapped along with the other field:

```
{
   "id":1,
   "firstName":"John",
   "lastName":"Doe"
}
```

6.4. @JsonView

@JsonView is used to *indicate the View* in which the property will be included for serialization/deserialization.

An example will show exactly how that works – we'll use *@JsonView* to serialize an instance of *Item* entity.

Let's start with the views:

```
public class Views {
   public static class Public {}
   public static class Internal extends Public {}
}
```

And now here's the *Item* entity, using the views:

```
public class Item {
   @JsonView(Views.Public.class)
   public int id;

   @JsonView(Views.Public.class)
   public String itemName;

   @JsonView(Views.Internal.class)
   public String ownerName;
}
```

Finally – the full test:

```
@Test
public void whenSerializingUsingJsonView_thenCorrect()
  throws JsonProcessingException {
    Item item = new Item(2, “book”, “John”);

    String result = new ObjectMapper().writerWithView(Views.Public.class)
                                .writeValueAsString(item);

    assertThat(result, containsString(“book”));
    assertThat(result, containsString(“2”));
    assertThat(result, not(containsString(“John”)));
}
```

6.5. @JsonManagedReference, @JsonBackReference

The *@JsonManagedReference* and *@JsonBackReference* annotations are used to *handle parent/child relationships*, and work around loops.

In the following example – we use *@JsonManagedReference* and *@JsonBackReference* to serialize our *ItemWithRef* entity:

```
public class ItemWithRef {
    public int id;
    public String itemName;

    @JsonManagedReference
    public UserWithRef owner;
}
```

Our *UserWithRef* entity:

```
public class UserWithRef {
    public int id;
    public String name;

    @JsonBackReference
    public List<ItemWithRef> userItems;
}
```

And the test:

```
@Test
public void whenSerializingUsingJacksonReferenceAnnotation_thenCorrect()
  throws JsonProcessingException {
    UserWithRef user = new UserWithRef(1, “John”);
    ItemWithRef item = new ItemWithRef(2, “book”, user);
    user.addItem(item);

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

    assertThat(result, containsString(“book”));
    assertThat(result, containsString(“John”));
    assertThat(result, not(containsString(“userItems”)));
}
```

6.6. @JsonIdentityInfo

@JsonIdentityInfo is used to indicate that Object Identity is to be used when serializing/deserializing values – for instance to deal with infinite recursion type of problems.

In the following example – we have an *ItemWithIdentity* entity with *a bidirectional realtionship* with the *UserWithIdentity* entity:

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = “id”)
public class ItemWithIdentity {
    public int id;
    public String itemName;
    public UserWithIdentity owner;
}
```

And the *UserWithIdentity* entity:

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")
public class UserWithIdentity {
    public int id;
    public String name;
    public List<ItemWithIdentity> userItems;
}
```

Now, let's see how the infinite recursion problem is handled:

```
@Test
public void whenSerializingUsingJsonIdentityInfo_thenCorrect()
  throws JsonProcessingException {
    UserWithIdentity user = new UserWithIdentity(1, "John");
    ItemWithIdentity item = new ItemWithIdentity(2, "book", user);
    user.addItem(item);

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

    assertThat(result, containsString("book"));
    assertThat(result, containsString("John"));
    assertThat(result, containsString("userItems"));
}
```

Here's the full output of the serialized item and user:

```
{
    "id": 2,
    "itemName": "book",
    "owner": {
        "id": 1,
        "name": "John",
        "userItems": [
            2
        ]
    }
}
```

6.7. @JsonFilter

The *@JsonFilter* annotation *specifies a filter to be used during serialization*.

Let's take a look at an example; first, we define the entity and we point to the filter:

```
@JsonFilter("myFilter")
public class BeanWithFilter {
   public int id;
   public String name;
}
```

Now, in the full test, we define the filter – which excludes all other properties except *name* from serialization:

```
@Test
public void whenSerializingUsingJsonFilter_thenCorrect()
  throws JsonProcessingException {
   BeanWithFilter bean = new BeanWithFilter(1, "My bean");

   FilterProvider filters =
     new SimpleFilterProvider().addFilter("myFilter",
     SimpleBeanPropertyFilter.filterOutAllExcept("name"));

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

   assertThat(result, containsString("My bean"));
   assertThat(result, not(containsString("id")));
}
```

7. Custom Jackson Annotation

Next – let's see how to create *a custom Jackson annotation*; we can make use of the @*JacksonAnnotationsInside* annotation – as in the following example:

```
@Retention(RetentionPolicy.RUNTIME)
   @JacksonAnnotationsInside
   @JsonInclude(Include.NON_NULL)
   @JsonPropertyOrder({ "name", "id", "dateCreated" })
   public @interface CustomAnnotation {}
```

Now, if we use the new annotation on an entity:

```
@CustomAnnotation
public class BeanWithCustomAnnotation {
   public int id;
   public String name;
   public Date dateCreated;
}
```

We can see how it does combine the existing annotations into a simpler, custom one that we can use as a shorthand:

```
@Test
public void whenSerializingUsingCustomAnnotation_thenCorrect()
  throws JsonProcessingException {
   BeanWithCustomAnnotation bean =
     new BeanWithCustomAnnotation(1, "My bean", null);

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

   assertThat(result, containsString("My bean"));
   assertThat(result, containsString("1"));
   assertThat(result, not(containsString("dateCreated")));
}
```

The output of the serialization process:

```
{
   “name”:”My bean”,
   “id”:1
}
```

8. Jackson MixIn Annotations

Next – let’s see how to use Jackson MixIn annotations.

Let’s use the MixIn annotations to – for example – ignore properties of type *String*:

```
public class User {
   public int id;
   public String name;
}

@JsonIgnoreType
public class MyMixInForString {}
```

Let’s see this in action:

```
@Test
public void whenSerializingUsingMixInAnnotation_thenCorrect()
  throws JsonProcessingException {
   User user = new User(1, “John”);
   String result = new ObjectMapper().writeValueAsString(user);
   assertThat(result, containsString(“John”));

   ObjectMapper mapper = new ObjectMapper();
   mapper.addMixInAnnotations(String.class, MyMixInForString.class);
   result = mapper.writeValueAsString(user);
   assertThat(result, not(containsString(“John”)));
}
```

9. Disable Jackson Annotation

Finally – let's see how we can *disable all Jackson annotations*.We can do this by disabling the *MapperFeature.USE_ANNOTATIONS* as in the following example:

```
@JsonInclude(Include.NON_NULL)
@JsonPropertyOrder({ "name", "id" })
public class MyBean {
   public int id;
   public String name;
}
```

Now, after disabling annotations, these should have no effect and the defaults of the library should apply:

```
@Test
public void whenDisablingAllAnnotations_thenAllDisabled()
  throws JsonProcessingException, IOException {
   MyBean bean = new MyBean(1, null);

   ObjectMapper mapper = new ObjectMapper();
   mapper.disable(MapperFeature.USE_ANNOTATIONS);
   String result = mapper.writeValueAsString(bean);

   assertThat(result, containsString("1"));
   assertThat(result, containsString("name"));
}
```

The result of serialization before disabling annotations:

```
{"id":1}
```

The result of serialization after disabling annotations:

```
{
   “id”:1,
   “name”:null
}
```

10. Conclusion

This section has been a deep-dive into Jackson annotations, just scratching the surface of the kind of flexibility you can get using them properly.

CHAPTER 2

Do JSON with Jackson

2: JACKSON IGNORE PROPERTIES ON MARSHALLING

1. Overview

This section will show how to *ignore certain fields when serializing an object to JSON* using Jackson 2.x.

This is very useful when the Jackson defaults aren't enough and we need to control exactly what gets serialized to JSON – and there are several ways to ignore properties.

2. Ignore fields at the **class** level

We can ignore specific fields at the **class** level, using *the @JsonIgnoreProperties annotation and specifying the fields by name*:

```
@JsonIgnoreProperties(value = { "intValue" })
public class MyDto {

    private String stringValue;
    private int intValue;
    private boolean booleanValue;

    public MyDto() {
        super();
    }

    // standard setters and getters are not shown
}
```

We can now test that, after the object is written to json, the field is indeed not part of the output:

```
@Test
public void givenFieldIsIgnoredByName_whenDtoIsSerialized_thenCorrect()
  throws JsonParseException, IOException {
    ObjectMapper mapper = new ObjectMapper();
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("intValue")));
}
```

3. Ignore field at the field level

We can also ignore a field directly via *the @JsonIgnore annotation directly on the field*:

```
public class MyDto {

    private String stringValue;
    @JsonIgnore
    private int intValue;
    private boolean booleanValue;

    public MyDto() {
        super();
    }


    // standard setters and getters are not shown
}
```

We can now test that the *intValue* field is indeed not part of the serialized json output:

```
@Test
public void givenFieldIsIgnoredDirectly_whenDtoIsSerialized_thenCorrect()
  throws JsonParseException, IOException {
    ObjectMapper mapper = new ObjectMapper();
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("intValue")));
}
```

4. Ignore all fields by type

Finally, we can *ignore all fields of a specified type, using the @JsonIgnoreType anotation*. If we control the type, then we can annotate the **class** directly:

```
@JsonIgnoreType
public class SomeType { ... }
```

More often than not however, we don't have control of the **class** itself; in this case, we can make good use of Jackson mixins.

First, we define a MixIn for the type we'd like to ignore, and annotate that with @ *JsonIgnoreType* instead:

```
@JsonIgnoreType
public class MyMixInForString {
    //
}
```

Then we register that mixin to replace (and ignore) all *String* types during marshalling:

```
mapper.addMixInAnnotations(String.class, MyMixInForString.class);
```

At this point, all Strings will be ignored instead of marshalled to JSON:

```
@Test
public final void givenFieldTypeIsIgnored_whenDtoIsSerialized_thenCorrect()
  throws JsonParseException, IOException {
    ObjectMapper mapper = new ObjectMapper();
    mapper.addMixInAnnotations(String.class, MyMixInForString.class);
    MyDto dtoObject = new MyDto();
    dtoObject.setBooleanValue(true);

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

5. Ignore fields using Filters

Finally, we can also use *Filters to ignore specific fields* in Jackson. First, we need to define the filter on the java object:

```
@JsonFilter("myFilter")
public class MyDtoWithFilter { ... }
```

Then, we define a simple filter that will ignore the *intValue* field:

```
SimpleBeanPropertyFilter theFilter = SimpleBeanPropertyFilter.serializeAllExcept("intValue");
FilterProvider filters = new SimpleFilterProvider().addFilter("myFilter", theFilter);
```

Now we can serialize the object and make sure that the *intValue* field is not present in the JSON output:

```
@Test
public final void givenTypeHasFilterThatIgnoresFieldByName_whenDtoIsSerialized_thenCorrect()
  throws JsonParseException, IOException {
    ObjectMapper mapper = new ObjectMapper();
    SimpleBeanPropertyFilter theFilter = SimpleBeanPropertyFilter.serializeAllExcept("intValue");
    FilterProvider filters = new SimpleFilterProvider().addFilter("myFilter", theFilter);

    MyDtoWithFilter dtoObject = new MyDtoWithFilter();
    String dtoAsString = mapper.writer(filters).writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("intValue")));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, containsString("stringValue"));
    System.out.println(dtoAsString);
}
```

6. Conclusion

The section illustrated how to ignore fields on serialization – first by name, then directly, and finally – we ignored the entire java type with MixIns and we use filters for more control of the output.

CHAPTER 3

Do JSON with Jackson

3: IGNORE NULL FIELDS WITH JACKSON

1. Overview

This quick section is going to cover how to set up *Jackson 2 to ignore null fields when serializing* a java **class**.

2. Ignore Null Fields on the Class

Jackson allows controlling this behavior at either the **class** level:

```
@JsonInclude(Include.NON_NULL)
public class MyDto { ... }
```

Or – more granularity – at the field level:

```
public class MyDto {

    @JsonInclude(Include.NON_NULL)
    private String stringValue;

    private int intValue;

    // standard getters and setters
}
```

Now, we should be able to test that null values are indeed not part of the final JSON output:

```
@Test
public void givenNullsIgnoredOnClass_whenWritingObjectWithNullField_thenIgnored()
  throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

3. Ignore Null Fields Globally

Jackson also allows configuring this behavior globally on the *ObjectMapper*:

```
mapper.setSerializationInclusion(Include.NON_NULL);
```

Now any null field in any **class** serialized through this mapper is going to be ignored:

```
@Test
public void givenNullsIgnoredGlobally_whenWritingObjectWithNullField_thenIgnored()
  throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    mapper.setSerializationInclusion(Include.NON_NULL);
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

4. Conclusion

Ignoring null fields is such a common Jackson configuration because it's often the case that we need to have better control over the JSON output. This section shows how to do that for **class**es – there are however more advanced use-cases.

CHAPTER 4

Do JSON with Jackson

4: JACKSON – CHANGE NAME OF FIELD

1. Overview

This quick section illustrates how to *change the name of a field to map to another json property* on serialization.

2. Change Name of Field for Serialization

Working with a simple entity:

```
public class MyDto {
    private String stringValue;

    public MyDto() {
        super();
    }

    public String getStringValue() {
        return stringValue;
    }

    public void setStringValue(String stringValue) {
        this.stringValue = stringValue;
    }
}
```

Serializing it will result in *the following JSON*:

```
{"stringValue":"some value"}
```

To *customize that output* so that, instead of *stringValue* we get – for example – *strValue*, we need to simply annotate the getter:

```
@JsonProperty("strVal")
public String getStringValue() {
   return stringValue;
}
```

Now, on serialization, *we will get the desired output*:

```
{"strValue":"some value"}
```

A simple unit test should verify the output is correct:

```
@Test
public void givenNameOfFieldIsChanged_whenSerializing_thenCorrect()
  throws JsonParseException, IOException {
   ObjectMapper mapper = new ObjectMapper();
   MyDtoFieldNameChanged dtoObject = new MyDtoFieldNameChanged();
   dtoObject.setStringValue("a");

   String dtoAsString = mapper.writeValueAsString(dtoObject);

   assertThat(dtoAsString, not(containsString("stringValue")));
   assertThat(dtoAsString, containsString("strVal"));
}
```

3. Conclusion

Marshalling an entity to adhere to a specific JSON format is a common task – and this section shows how to do is simply by using the *@JsonProperty* annotation.

CHAPTER 5

Do JSON with Jackson

5: JACKSON – MARSHALL STRING TO JSONNODE

1. Overview

This quick section will show how to *use Jackson 2 to convert a JSON String to a JsonNode* (com.fasterxml.jackson.databind.JsonNode).

2. Quick Parsing

Very simply, to parse the JSON String we only need an *ObjectMapper*:

```
@Test
public void whenParsingJsonStringIntoJsonNode_thenCorrect()
  throws JsonParseException, IOException {
    String jsonString = "{"k1":"v1","k2":"v2"}";

    ObjectMapper mapper = new ObjectMapper();
    JsonNode actualObj = mapper.readTree(jsonString);

    assertNotNull(actualObj);
}
```

3. Low Level Parsing

If, for some reason, you *need to go lower level* than that, the following example exposes the *JsonParser* responsible with the actual parsing of the String:

```
@Test
public void givenUsingLowLevelApi_whenParsingJsonStringIntoJsonNode_thenCorrect()
  throws JsonParseException, IOException {
    String jsonString = "{"k1":"v1","k2":"v2"}";

    ObjectMapper mapper = new ObjectMapper();
    JsonFactory factory = mapper.getFactory();
    JsonParser parser = factory.createParser(jsonString);
    JsonNode actualObj = mapper.readTree(parser);

    assertNotNull(actualObj);
}
```

4. Using the JsonNode

After the JSON is parsed into a JsonNode Object, we can *work with the Jackson JSON Tree Model*:

```
@Test
public void givenTheJsonNode_whenRetrievingDataFromId_thenCorrect()
  throws JsonParseException, IOException {
    String jsonString = "{"k1":"v1","k2":"v2"}";
    ObjectMapper mapper = new ObjectMapper();
    JsonNode actualObj = mapper.readTree(jsonString);

    // When
    JsonNode jsonNode1 = actualObj.get("k1");
    assertThat(jsonNode1.textValue(), equalTo("v1"));
}
```

5. Conclusion

This section illustrated how to parse JSON Strings into the Jackson JsonNode model to enable a structured processing of the JSON Object.

CHAPTER 6

Do JSON with Jackson

6: JACKSON UNMARSHALLING JSON WITH UNKNOWN PROPERTIES

1. Overview

In this section, we're going to take a look at the unmarshalling process with Jackson 2.x – specifically at *how to deal with JSONs with unknown properties*.

2. Unmarshall a JSON with additional/unknown fields

JSON input comes in all shapes and sizes – and most of the time, we need to map it to predefined java objects with a set number of fields. The goal is to simply *ignore any JSON properties that cannot be mapped to an existing java field*.

For example, say we need to unmarshall JSON to the following java entity:

```
public class MyDto {

    private String stringValue;
    private int intValue;
    private boolean booleanValue;

    public MyDto() {
        super();
    }

    // standard getters and setters and not included
}
```

2.1. UnrecognizedPropertyException on Unknown Fields

Trying to unmarshall a JSON with unknown properties to this simple Java Entity will lead to a *com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException*:

```
@Test(expected = UnrecognizedPropertyException.class)
public void givenJsonHasUnknownValues_whenDeserializing_thenException()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString =
        "{"stringValue":"a"," +
        ""intValue":1," +
        ""booleanValue":true," +
        ""stringValue2":"something"}";
    ObjectMapper mapper = new ObjectMapper();

    MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);

    assertNotNull(readValue);
}
```

This will fail with the following exception:

```
com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException:
Unrecognized field “stringValue2” (class org.baeldung.jackson.ignore.MyDto),
not marked as ignorable (3 known properties: “stringValue”, “booleanValue”, “intValue”])
```

2.2. Dealing with Unknown Fields on the ObjectMapper

We can now configure the full *ObjectMapper* to *ignore unknown properties in the JSON*:

```
new ObjectMapper().configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
```

We should then be able to read this kind of JSON into a predefined java entity:

```
@Test
public void
givenJsonHasUnknownValuesButJacksonIsIgnoringUnknowns_whenDeserializing_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString =
        “{“stringValue”:”a”,” +
        “”intValue”:1,” +
        “”booleanValue”:true,” +
        “”stringValue2”:”something”}”;
    ObjectMapper mapper =
      new ObjectMapper().configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);

    MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);

    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo(“a”));
    assertThat(readValue.isBooleanValue(), equalTo(true));
    assertThat(readValue.getIntValue(), equalTo(1));
}
```

2.3. Dealing with Unknown Fields on the **Class**

We can also mark a single **class** as accepting unknown fields, instead of the entire Jackson *ObjectMapper*:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class MyDtoIgnoreUnknown { ... }
```

Now, we should be able to test the same behavior as before – unknown fields are simply ignored and only known fields are mapped:

```
@Test
public void givenJsonHasUnknownValuesButIgnoredOnClass_whenDeserializing_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString =
        "{"stringValue":"a"," +
        ""intValue":1," +
        ""booleanValue":true," +
        ""stringValue2":"something"}";
    ObjectMapper mapper = new ObjectMapper();

    MyDtoIgnoreUnknown readValue = mapper.readValue(jsonAsString, MyDtoIgnoreUnknown.class);

    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo("a"));
    assertThat(readValue.isBooleanValue(), equalTo(true));
    assertThat(readValue.getIntValue(), equalTo(1));
}
```

3. Unmarshall an incomplete JSON

Similarly to additional unknown fields, unmarshalling an incomplete JSON – a JSON that doesn't contain all the fields in the java **class** – is not a problem with Jackson:

```
@Test
public void givenNotAllFieldsHaveValuesInJson_whenDeserializingAJsonToAClass_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString = "{"stringValue":"a","booleanValue":true}";
    ObjectMapper mapper = new ObjectMapper();

    MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);

    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo("a"));
    assertThat(readValue.isBooleanValue(), equalTo(true));
}
```

4. Conclusion

This section covered deserializing a JSON with additional, unknown properties, using Jackson. This is one of the most common things to configure when working with Jackson, since it's often the case to map JSONs of external REST APIs to an internal java representation of the entities of the API.