

Custom JSON Deserialization with Jackson

by A. N. M. Bazlur Rahman  · Dec. 17, 15 · Integration Zone · Tutorial

Heads up...this article is old!

Technology moves quickly and this article was published **2 years ago**. Some or all of its contents may be outdated.

SnapLogic is the leading self-service enterprise-grade integration platform. Download the Gartner iPaaS Magic Quadrant Report to learn more or play around on the platform, risk free, for 30 days.

We consume a REST API as a JSON format and then unmarshal it to a POJO. Jackson's `org.codehaus.jackson.map.ObjectMapper` “just works” out of the box and we really don't do anything in most cases. But sometimes we need a custom dserializer to fulfill our custom needs and this tutorial will guide you through the process of creating your own custom dserializer.

Let's say we have following entities:

```
1  public class User {
2      private Long id;
3      private String name;
4      private String email;
5
6      public Long getId() {
7          return id;
8      }
9
10     public User setId(Long id) {
11         this.id = id;
12         return this;
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public User setName(String name) {
20         this.name = name;
21         return this;
22     }
23
24     public String getEmail() {
25         return email;
26     }
27
28     public User setEmail(String email) {
29         this.email = email;
30         return this;
31     }
32
33     @Override
34     public String toString() {
35         final StringBuffer sb = new StringBuffer("User{");
36         sb.append("id=").append(id);
37         sb.append(", name=").append(name).append('\n');
38         sb.append(", email=").append(email).append('\n');
39         sb.append('}');
```

```
40 return sb.toString();
41 }
42 }
```

```
1 public class Program {
2     private Long id;
3     private String name;
4     private User createdBy;
5     private String contents;
6
7     public Program(Long id, String name, String contents, User createdBy) {
8         this.id = id;
9         this.name = name;
10        this.contents = contents;
11        this.createdBy = createdBy;
12    }
13
14    public Program() {}
15
16    public Long getId() {
17        return id;
18    }
19
20    public Program setId(Long id) {
21        this.id = id;
22        return this;
23    }
24
25    public String getName() {
26        return name;
27    }
28
29    public Program setName(String name) {
30        this.name = name;
31        return this;
32    }
33
34    public User getCreatedBy() {
35        return createdBy;
36    }
37
38    public Program setCreatedBy(User createdBy) {
39        this.createdBy = createdBy;
40        return this;
41    }
42
43    public String getContents() {
44        return contents;
45    }
46
47    public Program setContents(String contents) {
48        this.contents = contents;
49        return this;
50    }
51
52    @Override
53    public String toString() {
54        final StringBuffer sb = new StringBuffer("Program{");
55        sb.append("id=").append(id);
56        sb.append(", name=").append(name).append('\n');
57        sb.append(", createdBy=").append(createdBy);
58        sb.append(", contents=").append(contents).append('\n');
59        sb.append('}');
60        return sb.toString();
61    }
62 }
```

```
61     }
62 }
```

Let's serialize/marshal an object first:

```
1  User user = new User();
2  user.setId(1L);
3  user.setEmail("example@example.com");
4  user.setName("Bazlur Rahman");
5
6  Program program = new Program();
7  program.setId(1L);
8  program.setName("Program @# 1");
9  program.setCreatedBy(user);
10 program.setContents("Some contents");
11
12 ObjectMapper objectMapper = new ObjectMapper();
13
14 final String json = objectMapper.writeValueAsString(program);
15 System.out.println(json);
```

The above code will produce the following JSON:

```
1  {
2  "id": 1,
3  "name": "Program @# 1",
4  "createdBy": {
5  "id": 1,
6  "name": "Bazlur Rahman",
7  "email": "example@example.com"
8  },
9  "contents": "Some contents"
10 }
```

Now can do the opposite very easily. If we have this JSON, we can unmarshal to a program object using ObjectMapper as following:

```
String jsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"createdBy\":{\"id\":1,\"name\":\"Bazlur Rahman\",\"email\":\"example@example.com\"},\"contents\":\"Some contents\"}";
1
2
3 final Program program1 = objectMapper.readValue(jsonString, Program.class);
4 System.out.println(program1);
```

Now let's say, this is not the real case, we are going to have a different JSON from an API which doesn't match with our program class:

```
1  {
2  "id": 1,
3  "name": "Program @# 1",
4  "ownerId": 1
5  "contents": "Some contents"
6  }
```

Look at the json string, it has a different field that is `ownerId`.

Now if you want to deserialize this JSON as we did earlier, you will have exceptions.

There are two ways to handle this. First, you can ignore the unknown fields and write a custom deserializer. Second, you can learn how to improve communication between product management & developer teams.



The 2018 Guide to DevOps

Best Practices for Continuous Integration, Continuous Delivery & Sprint Planning

Discover the Continuous Delivery Anti-Patterns You Must Avoid

Learn how to Improve Communication Between Product Management & Developer Teams

Ignore the Unknown Fields

[Download My Free PDF](#)

Ignore the `ownerId`. Add following annotation in the Program class

```
1 @JsonIgnoreProperties(ignoreUnknown = true)
2 public class Program {}
```

Write Custom Deserializer

But there are cases when you actually need this `ownerId` field. Lets say you want to relate it as an id of the user class.

In such case, you need to write a custom deserializer:

```

1  import com.fasterxml.jackson.core.JsonParser;
2  import com.fasterxml.jackson.core.JsonProcessingException;
3  import com.fasterxml.jackson.core.ObjectCodec;
4  import com.fasterxml.jackson.databind.DeserializationContext;
5  import com.fasterxml.jackson.databind.JsonDeserializer;
6  import com.fasterxml.jackson.databind.JsonNode;
7
8  import java.io.IOException;
9
10 public class ProgramDeserializer extends JsonDeserializer<Program> {
11     @Override
12     public Program deserialize(JsonParser jp, DeserializationContext ctxt) throws IOException, JsonProcessingException {
13         ObjectCodec oc = jp.getCodec();
14         JsonNode node = oc.readTree(jp);
15
16         final Long id = node.get("id").asLong();
17         final String name = node.get("name").asText();
18         final String contents = node.get("contents").asText();
19         final long ownerId = node.get("ownerId").asLong();
20
21         User user = new User();
22         user.setId(ownerId);
23
24         return new Program(id, name, contents, user);
25     }
26 }

```

As you can see, first you have to access the `JsonNode` from the `JsonParser`. And then you can easily extract information from a `JsonNode` using `get` method. and you have to be make sure about the field name. It should be the exact name, spelling mistake will cause exceptions.

And finally you have to register your `ProgramDeserializer` to the `ObjectMapper`.

```

1  ObjectMapper mapper = new ObjectMapper();
2  SimpleModule module = new SimpleModule();
3  module.addDeserializer(Program.class, new ProgramDeserializer());
4
5  mapper.registerModule(module);
6
7  String newJsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"ownerId\":1,\"contents\":\"Some contents\"}";
8  final Program program2 = mapper.readValue(newJsonString, Program.class);

```

Alternatively you can use annotation to register the deserializer directly:

```

1  @JsonDeserialize(using = ProgramDeserializer.<b>class</b>)
2  public class Program {}

```

Full source code can be found in : <https://github.com/rokon12/json-deserializer>

With SnapLogic's integration platform you can save millions of dollars, increase integrator productivity by 5X, and reduce integration time to value by 90%. Sign up for our risk-free 30-day trial!

Like This Article? Read More From DZone



Jackson Mixin to the Rescue



JSON processing using Jackson Java JSON Processor



Dynamically Filter JSON With Jackson and Squiggly




Free DZone Refcard RESTful API Lifecycle Management


Topics: [JAVA](#) , [JSON](#) , [JACKSONJSON](#) , [JACKSON](#) , [DESERIALIZATION](#)

Integration Partner Resources

The CRM Integration Guide: 8 Things Experts Are Considering in Their API Strategy
Cloud Elements

 How event streaming can benefit API design.
Capital One

 Modernizing Application Architectures with Microservices and APIs
CA Technologies

 A Comprehensive Guide to the Enterprise Integration Cloud
SnapLogic
