9-Jackson, Deserialization & Serialization

Author: Vincent Lau

Learning Objectives

- Understand the framework Jackson for deserialization and serialization
- Understand how ObjectMapper for testing deserialization and serialization process

Introduction

`Serialization` and `deserialization` are essential concepts in web applications for converting data between its in-memory object representation and formats suitable for storage, transport, or exchange, such as JSON or XML.

In the context of a Spring Boot application, serialization and deserialization are often used to convert Java objects to JSON (or other formats) for HTTP requests and responses. Here's an introduction to serialization and deserialization and how to implement them in a Spring Boot application:

Serialization

Definition: Serialization is the process of converting an in-memory Java object into a byte stream or a format that can be stored in a file or sent over the network. Serialization is commonly used when you need to persist the state of an object or transmit it across different systems.

Use Cases: Serialization is used when saving objects to databases, caching, messaging, or creating APIs that send object data as a response.

Deserialization

Definition: Deserialization is the process of converting a byte stream or a serialized format back into an in-memory Java object. It's the reverse of serialization and is used to reconstruct objects from previously serialized data.

Use Cases: Deserialization is used when retrieving objects from storage, parsing incoming data from network requests, or reading from a message queue.

Implementing Serialization and Deserialization in a Spring Boot Application:

In a Spring Boot application, you can easily implement serialization and deserialization by using libraries and annotations. Here's how to do it:

1. Choose a Serialization Format:
2. Decide on the serialization format you want to use. JSON is a common choice for web APIs due to its simplicity and human-readability. Spring Boot supports JSON serialization and deserialization out of the box, but you can also work with other formats like XML or Protocol Buffers if needed.
3. Create Java Model Classes:
4. Define Java model classes that represent your data. These classes should have appropriate getters and setters for the fields you want to serialize and deserialize.
5. Example:

```
public class User {
    private String id;
    private String username;
    private String email;

    // getters and setters
}
```

Jackson

Spring Boot relies on the Jackson library for JSON serialization and deserialization by default. Jackson is highly configurable and can handle complex object mappings. Ensure that Jackson is included as a dependency in your project.

Maven Dependency

Add the following dependencies to your `pom.xml` (if using Maven):

```xml
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.5</version> <!-- Use the appropriate version -->
</dependency>
```

Object Mapper

Serialize Objects to JSON

To serialize Java objects to JSON in a Spring Boot application, you can use Jackson's `ObjectMapper`:

```java
ObjectMapper objectMapper = new ObjectMapper();
String json = objectMapper.writeValueAsString(user); // Serialize the User object to JSON
```

Deserialize JSON to Objects

To deserialize JSON back into Java objects, you can use Jackson's `ObjectMapper`:

```java
String json = "{\"id\":\"123\",\"username\":\"john\",\"email\":\"john@example.com\"}";
User user = objectMapper.readValue(json, User.class); // Deserialize JSON to a User object
```

Auto-Deserialization

Spring Boot Controllers

In a Spring Boot application, when you create RESTful APIs using `@RestController` and `@RequestMapping` annotations, Spring Boot automatically serializes and deserializes objects to and from JSON for you.

For example, in a controller method, you can return a Java object, and Spring Boot will automatically serialize it to JSON for the HTTP response.

```
@RestController
public class UserController {
    @GetMapping("/user/{id}")
    public User getUser(@PathVariable String id) {
        // Fetch User object from the database and return it
        User user = userRepository.findById(id);
        return user; // Spring Boot will serialize it to JSON
    }
}
```

RestTemplate

- `RestTemplate` automatically detects the `Content-Type` header in the HTTP response and determines the appropriate message converter to use for deserialization. By default, it uses the `MappingJackson2HttpMessageConverter` for JSON responses, which relies on Jackson for deserialization.
- The response body, which is in JSON format, is passed to the `MappingJackson2HttpMessageConverter` for deserialization.
- Jackson Deserialization
  - Jackson, the default JSON processing library in Spring, is responsible for deserializing the JSON data into Java objects.
  - Jackson uses reflection and Java object mapping to create instances of Java classes that correspond to the JSON data structure.
  - The JSON property names in the response are matched to the field names or getter/setter methods in the Java class. Jackson uses these mappings to populate the Java objects.
- Java Objects:
  - After deserialization, you receive Java objects that represent the data retrieved from the API. You can work with these objects as you would with any other Java objects.

Here's an example of how you might use `RestTemplate` to make an HTTP GET request and deserialize the response into Java objects:

```
```

```
RestTemplate restTemplate = new RestTemplate();
String apiUrl = "https://api.example.com/users";
UserResponse userResponse = restTemplate.getForObject(apiUrl,
UserResponse.class);
```

In this example, `UserResponse` is a Java class that represents the expected structure of the JSON response from the API. `RestTemplate` retrieves the JSON response, and Jackson handles the deserialization, populating the `userResponse` object with the data from the response.

Annotations

Here are examples of some of the commonly used annotations from the `com.fasterxml.jackson.databind.annotation` package in the Jackson library.

@JsonSerialize & @JsonDeserialize

- `@JsonSerialize` allows you to specify a custom serializer for a field or class during serialization.
- `@JsonDeserialize` allows you to specify a custom deserializer during deserialization.

In this example, `CustomDateSerializer` is a custom serializer, and `CustomDateDeserializer` is a custom deserializer for the `date` field.

```
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;

@JsonSerialize(using = CustomDateSerializer.class)
@JsonDeserialize(using = CustomDateDeserializer.class)
public class CustomDateExample {
    private Date date;

    // getters and setters
}
```

We create a custom serializer to format dates as ISO 8601 strings. The `CustomDateSerializer` class extends `JsonSerializer<Date>`. It overrides the `serialize` method to format the date as an ISO 8601 string using the specified date format.

```
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
4
5  import com.fasterxml.jackson.core.JsonGenerator;
6  import com.fasterxml.jackson.databind.JsonSerializer;
7  import com.fasterxml.jackson.databind.SerializerProvider;
8
9  public class CustomDateSerializer extends JsonSerializer<Date> {
10     private SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
   dd'T'HH:mm:ss.SSSZ");
11
12     @Override
13     public void serialize(Date date, JsonGenerator gen, SerializerProvider
   serializers) throws IOException {
14         String formattedDate = dateFormat.format(date);
15         gen.writeString(formattedDate);
16     }
17 }
```

A custom date deserializer allows you to specify how dates should be parsed when deserializing JSON into Java objects. In this example, we create a custom deserializer to parse ISO 8601 date strings into `Date` objects. The `CustomDateDeserializer` class extends `JsonDeserializer<Date>`. It overrides the `deserialize` method to parse ISO 8601 date strings into `Date` objects using the specified date format.

```
1  import java.io.IOException;
2  import java.text.ParseException;
3  import java.text.SimpleDateFormat;
4  import java.util.Date;
5
6  import com.fasterxml.jackson.core.JsonParser;
7  import com.fasterxml.jackson.databind.DeserializationContext;
8  import com.fasterxml.jackson.databind.JsonDeserializer;
9
10 public class CustomDateDeserializer extends JsonDeserializer<Date> {
11     private SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
   dd'T'HH:mm:ss.SSSZ");
12
13     @Override
14     public Date deserialize(JsonParser p, DeserializationContext ctxt) throws
   IOException {
15         String dateStr = p.getValueAsString();
16         try {
17             return dateFormat.parse(dateStr);
18         } catch (ParseException e) {
19             throw new IOException("Error parsing date: " + dateStr, e);
```

```
        }
    }
}
```

@JsonFormat

`@JsonFormat` is used to control the format of date/time values during serialization and deserialization.

In this example, the `date` field is formatted as a string in the "yyyy-MM-dd'T'HH:mm:ss.SSSZ" format.

```
import com.fasterxml.jackson.annotation.JsonFormat;

public class DateFormatExample {
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss.SSSZ")
    private Date date;

    // getters and setters
}
```

@JsonProperty & @JsonAlias

`@JsonProperty` is used to specify the name of a field in JSON, and `@JsonAlias` specifies multiple possible names for a field during deserialization.

The `userId` field is mapped to the JSON property "user_id," and the `name` field can be deserialized from JSON properties "username" or "userName."

```
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonAlias;

public class PropertyNameExample {
    @JsonProperty("user_id")
    private String userId;

    @JsonAlias({"username", "userName"})
    private String name;

    // getters and setters
}
```

@JsonNaming

`@JsonNaming` allows you to specify a custom naming strategy.

In this example, `SnakeCaseStrategy` is a naming strategy that you can implement to define your own naming conventions.

```
{
  "id": 1,
  "user_name": "john_doe",
  "email_address": "john@example.com"
}
```

```
import com.fasterxml.jackson.databind.PropertyNamingStrategy;
import com.fasterxml.jackson.databind.annotation.JsonNaming;

@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
public class NamingBean {
    private long id;
    private String userName;
    private String emailAddress;

    // constructor, getters and setters
}
```

@JsonIgnore

The `@JsonIgnore` annotation is used to exclude a specific field from serialization and deserialization. It's particularly useful when you want to omit certain fields from the JSON representation.

In this example, the `password` field is annotated with `@JsonIgnore`, which means that it will not be included in the JSON representation when serializing or deserializing a `User` object.

```
import com.fasterxml.jackson.annotation.JsonIgnore;

public class User {
    private String id;
    private String username;

    @JsonIgnore
    private String password;

```

```
    // getters and setters
}
```

@JsonIgnoreProperties

- The `@JsonIgnoreProperties` annotation is used to **exclude multiple fields** from serialization and deserialization for an entire class. You can specify one or more field names to be ignored.

```
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties({ "internalField1", "internalField2" })
public class Employee {
    private String name;
    private int age;
    private String internalField1;
    private String internalField2;

    // getters and setters
}
```

- The `@JsonIgnoreProperties` annotation, when used with the `ignoreUnknown` attribute set to `true`, allows you to gracefully handle JSON data that contains extra, unrecognized fields during deserialization.

Here's an example of how to use `@JsonIgnoreProperties(ignoreUnknown = true)`:

```
{
  "id": 1,
  "username": "john_doe",
  "email": "john@example.com",
  "extraField1": "value1",
  "extraField2": "value2"
}
```

You want to deserialize this JSON into a Java object while ignoring the extra fields that are not part of the `User` class. Here's how you can achieve this:

We use `@JsonIgnoreProperties(ignoreUnknown = true)` to instruct Jackson to ignore any extra fields in the JSON data during deserialization.

```java
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class User {
    private Long id;
    private String username;
    private String email;

    // getters and setters
}
```

```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class Main {
    public static void main(String[] args) throws Exception {
        String json = "{\"id\":1,\"username\":\"john_doe\",\"email\":\"john@example.com\",\"extraField1\":\"value1\",\"extraField2\":\"value2\"}";

        ObjectMapper objectMapper = new ObjectMapper();
        User user = objectMapper.readValue(json, User.class);

        System.out.println(user.getUsername()); // Outputs: john_doe
        System.out.println(user.getEmail());    // Outputs: john@example.com
        System.out.println(user.getId());       // Outputs: 1 (if getter is implemented)
    }
}
```

In this example, the JSON data contains two extra fields, "extraField1" and "extraField2," which are not part of the `User` class. However, because of the `@JsonIgnoreProperties(ignoreUnknown = true)` annotation, Jackson ignores these extra fields during deserialization and successfully creates a `User` object with the recognized fields.

@JsonSetter

`@JsonSetter`:

The `@JsonSetter` annotation is used to customize the name of the setter method and associate it with a specific JSON property. It allows you to map a JSON property to a different setter method name.

```java
import com.fasterxml.jackson.annotation.JsonSetter;

public class User {

    private String username;

    @JsonSetter("user_name")
    public void setUserName(String username) {
        this.username = username;
    }

    // Getter and other methods
}
```

In this example, the `setUserName` method is annotated with `@JsonSetter("user_name")`, which means that during deserialization, Jackson will map the JSON property "user_name" to the `setUserName` method.

@JsonGetter

The `@JsonGetter` annotation is used to customize the name of the getter method and associate it with a specific JSON property. It allows you to map a JSON property to a different getter method name.
The `getUserName` method is annotated with `@JsonGetter("user_name")`, which means that during serialization, Jackson will use the `getUserName` method to retrieve the value for the JSON property "user_name."

```java
import com.fasterxml.jackson.annotation.JsonGetter;

public class User {
    private String username;

    @JsonGetter("user_name")
    public String getUserName() {
        return username;
    }

    // Setter and other methods
}
```

In this example, Jackson uses the `@JsonSetter` and `@JsonGetter` annotations to map the JSON property "user_name" to the `setUserName` and `getUserName` methods, respectively, during deserialization and serialization.

```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class Main {
    public static void main(String[] args) throws Exception {
        String json = "{\"user_name\":\"john_doe\"}";

        ObjectMapper objectMapper = new ObjectMapper();
        User user = objectMapper.readValue(json, User.class);

        System.out.println(user.getUserName()); // Outputs: john_doe
    }
}
```

@JsonRawValue

`@JsonRawValue` is used to include a field's value in the JSON output as-is, without further serialization.

In this example, the `jsonField` value is treated as raw JSON data.

```java
import com.fasterxml.jackson.annotation.JsonRawValue;

public class RawValueExample {
    @JsonRawValue
    private String jsonField;

    // getters and setters
}
```

These are just a few examples of how you can use Jackson annotations to customize the serialization and deserialization of Java objects to and from JSON. Jackson provides a wide range of annotations to handle various scenarios and requirements when working with JSON data in your Java applications.

Summary

All the aboves demonstrate how to implement serialization and deserialization in a Spring Boot application, particularly for JSON data. You can adjust the configuration and use other serialization formats as needed for your specific use cases. Spring Boot's integration with libraries like Jackson simplifies the process of working with serialized data in web applications.