Serialization and Deserialization

In this lesson, we will see what serialization and deserialization of requests is.

We'll cover the following

What is serialization?

Serialization is the process of converting objects into a stream of data.

What is deserialization?

Deserialization is the process of converting a stream of data into objects.

The main purpose of ***serialization*** and ***deserialization*** is to persist the data and recreate whenever needed.

We have considered the `Rest Assured` library for making REST API calls. We will keep the scope within those capabilities of `Rest Assured` and the libraries it depends on.

As we keep learning about REST API automation and the data that is exchanged between client and server is of JSON format, we will learn how to serialize objects into a stream of JSON data and deserialize stream of data to objects that are exchanged between the REST web service.

`Rest Assured` can use the Jackson 2 library, GSON library or Jackson library for serialization and deserialization. The internal behavior of `io.restassured.mapper.ObjectMapper` is dependent on the library in the classpath.

Sample data for demonstration & understanding the data

We will use the Jackson 2 library for serialization and deserialization purposes, for which we will ensure the following dependency from here is added:

- **Maven**

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.3</version>
</dependency>
```

- **Gradle**

```
compile 'com.fasterxml.jackson.core jackson-databind:2.10.3'
```

Let us consider the class `Student` for demonstration purposes.

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

public class Student {

    @JsonProperty("id")
    private Long id;

    @JsonProperty("first_name")
    private String firstName;

    @JsonProperty("last_name")
    private String lastName;

    @JsonProperty("gender")
    private String gender;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
```

```
        return firstName;
    }


    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    @Override
    public String toString() {
        return String.format("Student [id=%s, firstName=%s, lastName=%s, gende
r=%s]", id, firstName, lastName, gender);
    }

}
```

The above class `Student` contains the following:

- `Fields` – ***id***, ***firstName***, ***lastName***, ***gender***
- `Getters` – for fetching the field values
- `Setters` – for setting the field values
- `toString()` – for printing the object

As we can see in the `Student` class, all the fields are annotated with `@JsonProperty`. The purpose of having that annotation over the fields is that, during the process of serialization to JSON or deserialization from JSON, we need to know what key the fields should be mapped to.

As per Java conventions, we tend to name all the field names in ***camel case***. The

As per Java conventions, we tend to name all the field names in ***camel case***. The Jackson library, by default, takes the field name as the key during the process of serialization and deserialization. However, sometimes, we may want to have a different key for the field during the process. For that purpose, we use `@JsonProperty` which overrides the default behavior and the `String` given in the annotation will be taken as the key for that field.

`@JsonProperty` can be annotated over ***setter*** methods also. If the annotation is given in both places, we get the following exception:

```
java.lang.IllegalStateException: Conflicting/ambiguous property name definitio
ns (implicit name 'firstName'): found multiple explicit names: [f_name, first_
name], but also implicit accessor: [method restassured.Student#getFirstName(0
 params)][visible=true,ignore=false,explicitName=false]
```

The Jackson library provides us with so many other annotations to help us override the default behavior of the library upon fields during the process of serialization and deserialization. To know more, please follow this link.

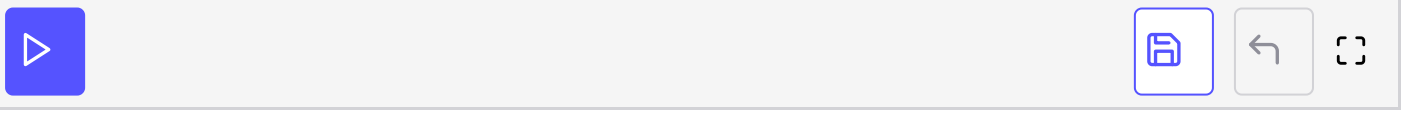
How to serialize an object?

Now, we will see how to set Java objects to request the body of an API so that `Rest Assured` can serialize the object into a stream of JSON data before making the API call.

```
import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;

import java.io.IOException;
import java.lang.reflect.Type;
import java.util.Arrays;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.Test;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.ObjectMapper;

import io.restassured.RestAssured;
import io.restassured.response.Response;

public class APIDemo {

        private static final Logger LOG = LoggerFactory.getLogger(APIDemo.class);

        private static final ObjectMapper MAPPER = new ObjectMapper();
```

```
	private Integer id;

	@Test
	public void serializationTest() throws IOException {

		// creating `Student` object
		Student student = new Student("Sam", "Bailey", "Female");

		// converting `Student` object to JSON string using `ObjectMapper`
		byte[] data = MAPPER.writeValueAsBytes(student);
		String json = MAPPER.writeValueAsString(student);

		LOG.info("serialization of `Student` class into JSON string using `ObjectMapper` =
		LOG.info("serialization of `Student` class into JSON string using `ObjectMapper` =

		// using `Student` object in body of `CreateStudent` API
		String url = "http://ezifyautomationlabs.com:6565/educative-rest/students";
		Response response = RestAssured.given()
				.contentType("application/json")
				.log().all(true)
				.accept("application/json")
				.body(student)
				.post(url)
				.andReturn();

		// validating the HTTP status code
		assertEquals(response.getStatusCode(), 201, "http status");

		// saving the `id` of the created `Student` to delete the same in cleanup method
		id = response.path("id");

		// validating whether the created `Student` id not null
		assertNotNull(id, "created student id is null");

		LOG.info("created student id => {}", id);
	}

	@AfterMethod
	public void deleteUser() {
		if (id != null) {
			String url = "http://ezifyautomationlabs.com:6565/educative-rest/students/
			Response response = RestAssured.given()
				.contentType("application/json")
					.accept("application/json")
					.pathParam("id", id)
					.delete(url);
			assertEquals(response.getStatusCode(), 204, "http status");
		}
	}

}

class Student {

	public Student() {
	}

	public Student(String firstName, String lastName, String gender) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.gender = gender;
```

```
        }

        Long id;

        @JsonProperty("first_name")
        String firstName;

        @JsonProperty("last_name")
        String lastName;

        String gender;

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

        public String getGender() {
                return gender;
        }

        public void setGender(String gender) {
                this.gender = gender;
        }

        @Override
        public String toString() {
                return String.format("Student [id=%s, firstName=%s, lastName=%s, gender=%s]", id,
        }

}
```



In the code above, we see the body of the POST API is the `Student` object. Based on `content-type` header, the body content is serialized. In our case, the `Student` object is serialized into a stream of JSON string data.

If the `content-type` is not set, we get the `415 Unsupported Media Type` HTTP status code, as the server won't be able to understand the request body format.

If we are setting the body with JSON string directly instead of the `Student` object, then it is inferred that the `content-type` is JSON and we won't get the `415 Unsupported Media Type` HTTP status code error.

How to deserialize into the object?

Next, we will see how to deserialize the response data stream into appropriate Java objects.

```
import java.io.IOException;
import java.lang.reflect.Type;
import java.util.Arrays;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.testng.annotations.Test;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;

import io.restassured.RestAssured;

public class APIDemo {

	private static final Logger LOG = LoggerFactory.getLogger(APIDemo.class);

	private static final ObjectMapper MAPPER = new ObjectMapper();

	@Test
	public void deserializationTest() throws IOException {

		String json = "{\"id\":100,\"gender\":\"Female\",\"first_name\":\"Sam\",\"last_nam

	Student student = MAPPER.readValue(json, Student.class);
		LOG.info("deserialization of JSON string into `Student` class => {}", student);

		String url = "http://ezifyautomationlabs.com:6565/educative-rest/students/{id}";
		Student studentA = RestAssured
			.given()
				.pathParam("id", "100")
				.get(url)
				.as(Student.class);
		LOG.info("deserialization of JSON string into class `Student` => {}", studentA);

		url = "http://ezifyautomationlabs.com:6565/educative-rest/students";
		Student[] studentsArray = RestAssured
				.get(url)
				.as(Student[].class);
		LOG.info("deserialization of JSON string into `Student[]` => {}", Arrays.deepToStr

	Type type = new TypeReference<List<Student>>() {}.getType();
```

```
                List<Student> students = RestAssured
                                .get(url)
                                .as(type);

                LOG.info("deserialization of JSON string into class with type parameter `List<Stud

        }

}

class Student {

        Long id;

        @JsonProperty("first_name")
        String firstName;

        @JsonProperty("last_name")
        String lastName;
        String gender;

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

        public String getGender() {
                return gender;
        }

        public void setGender(String gender) {
                this.gender = gender;
        }

        @Override
        public String toString() {
                return String.format("Student [id=%s, firstName=%s, lastName=%s, gender=%s]", id,
        }

}
```

In the code above sample, we can see how to:

- deserialize a JSON string into the `Student` object using `ObjectMapper`.
- deserialize response from the API into the `Student` object
- deserialize response from the API into the `Student[]` object
- deserialize response from the API into the `List<Student>` object. Since `List<T>` is a class with a type parameter, we need to use the following code, where we need to pass the class type:

  ```
  Type type = new TypeReference<List<Student>>() {}.getType();
  List<Student> students = RestAssured
              .get(url)
              .as(type);
  ```

  Otherwise, we can simply put `Student.class` in place of type.

Please note that the deserialization will fail if no matching field is found for the JSON key. To ignore the failure, we can annotate the class with `@JsonIgnoreProperties(ignoreUnknown = true)`.

There are more such annotations that we can use to override the default behavior of the underlying Jackson library using these annotations listed here.

In the next lesson, we will learn about creating `Specification` in `Rest Assured` that is used to construct requests and responses.