Serializing Reladomo Object Graphs

May 2017

Table of Contents

1. Introduction 1
2. Usage Guidelines 1
3. Serialization 2
4. Deserialization 3
5. Example Jackson/JAX-RS Serialization 3
6. Implementing Your Own Serialization/Deserialization 4
    6.1. Serialization 4
    6.2. Deserialization 4

1. Introduction

Serializing object graphs of any kind has some challenges including, the depth of the serialization and dealing with loops. Serializing a persistent object graph creates additional challenges, including efficient IO when reading/writing, proper designation of unit of work and so on. Reladomo's serialization utilities consist of a set of API that simplify and standardize the solutions to these issues.

2. Usage Guidelines

Remote API's can have a detrimental effect on the agility of a codebase if not designed appropriately. Avoid exposing your domain as a public API. Doing so will make changing your domain significantly harder and in some sense is no different than letting random callers read access to your database. Making a code change will require agreement for a large number of potentially unknown users. Instead of exposing your domain publicly, expose a well defined, narrow API that satisfies your requirements. Private API, that is, API that is used within a single codebase, does not have the same issue, so it may expose the domain.

The method signatures of your API should include the wrapper objects `Serialized` and `SerializedList`. For example:

```
@Path("/orderOne")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Serialized<Order> firstOrder()
{
    SerializationConfig config =
      SerializationConfig.shallowWithDefaultAttributes(
        OrderFinder.getFinderInstance());
    return
      new
 Serialized((OrderFinder.findOne(OrderFinder.orderId().eq(1))),
      config);
}
```

This is important for serialization because it allows you to fully specify (via `SerializationConfig`) how you want the graph to be serialized. It's also important for deserialization in method parameters to avoid doing unnecessary IO, potentially outside the unit of work that will start in the method body.

The core serialization framework is not tied to any particular implementation. It can be used to serialize to various formats, e.g. binary, xml, json, etc. There are Jackson and Gson example implementations that can used as is, or as a starting point for your own format. We recommend using the Jackson implementation for json/rest.

3. Serialization

When an object is serialized, it's the equivalent of detaching (in the Reladomo definition of detach) the object and serializing the result. The `Serialized` wrapper class allows the injection of a particular set of configurations for a response. For example, if you have three API methods and they each need to serialize the same type of result differently, the `Serialized` wrapper class enables that via `SerializationConfig`.

`SerializationConfig` is an immutable class with factory (builder) methods. You typically start with:

```
SerializationConfig config =
  SerializiztionConfig.shallowWithDefaultAttributes(
    OrderFinder.getFinderInstance());
```

and then use the `with` and `without` methods to create new instances of `SerializationConfig`. For example:

```
config = config.withDeepFetches(OrderFinder.orderStatus(),
  OrderFinder.items());
config = config.withAnnotatedMethods(SerialView.Shorter.class);
```

It's safe to store the instance for later use because it's immutable. The storage and reuse of `SerializationConfig` You can assign a name to a configuration and save it for later use with

```
public void saveOrOverwriteWithName(String name)
```

and `SerializationConfig.byName(String name)`.

You may annotate your domain methods (those implemented in your concrete classes) using the `@ReladomoSerialize` annotation. The annotation takes a list of classes that correspond to a the particular view. For example, you can define a set of views like this:

```
public class SerialView
{
    public static class Shorter {}
    public static class Longer extends Shorter {}
    public static class HandPicked {}
}
```

and then reference that on a method:

```
```

```
@ReladomoSerialize(serialViews = {SerialView.Shorter.class,
 SerialView.HandPicked.class})
public String getTrackedDescription()
{
    return this.getDescription()+" "+this.getTrackingId();
}
```

This method will only be serialized if the `SerializationConfig` has either `.withAnnotatedMethods(SerialView.Shorter.class` or `.withAnnotatedMethods(SerialView.HandPicked.class`.

The object tree is handled very similarly to the way deep fetching works in Reladomo. You can specify the navigation paths through the tree using the `.withDeepFetches` method on `SerializationConfig`.

4. Deserialization

Deserialization happens when method parameters are Reladomo objects, or one of the wrapper classes `Serialized` and `SerializedList`. The wrappers are again recommended, but for a different reason. A unit of work (transaction) does not encompass the deserialization, which often requires database lookups. For proper transactional enrollment, those lookup will likely be repeated in the method body if a wrapper class is not used. In contrast, a `Serialized` object will delay the lookups until the `getWrapped()` method is called.

The deserializaton protocol requires either a specifically typed object (e.g. `Serialized<Order>`) or meta data in the stream to specify the object class. We recommend having the meta data in the stream for simplicity. The meta data can also include a state, which specifies if the incoming object is to be considered in-memory (new), detached or deleted. Without state metadata, the state is considered to be detached or new and further determined by looking up the object in the database. The deleted state is particularly useful for sending inserts/updates/deletes in a single call for a list of objects.

Attributes that are not in the stream at all are considered to be unchanged (or default if the object is new). This allows for the more common "patch" implementation when persisting.

5. Example Jackson/JAX-RS Serialization

The example Jackson implementation in the `com.gs.reladomo.serial.jackson` package can be used to serialize and deserialize json. You'll need reladomo-serial.jar in your classpath. The implementation can be use with JAX-RS by following these steps:

- Use `com.gs.reladomo.serial.jaxrs.server.JacksonObjectMapperProvider` or write a similar class.
- In your server's resource config, register the mapper provider.
- In your resource (remote API), use `@Produces` and/or `@Consumes` `MediaType.APPLICATION_JSON`
  For example:

```
@Path("/orderOne")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Serialized<Order> firstOrder()
```

```
{
    SerializationConfig config =
      SerializationConfig.shallowWithDefaultAttributes(
        OrderFinder.getFinderInstance());
    return
      new
 Serialized((OrderFinder.findOne(OrderFinder.orderId().eq(1))),
        config);
}
```

6. Implementing Your Own Serialization/ Deserialization

6.1. Serialization

The Jackson and Gson implementations are good examples to follow.

Every serialization implementation requires a `ReladomoSerializationContext` and a `SerialWriter`. Typically, `ReladomoSerializationContext` is subclassed and the implementation's local context is added to the subclass. `ReladomoSerializationContext` has to be instantiated for every tree being serialized. It usually contains some sort of output stream, which is passed to the `SerialWriter`. The `SerialWriter` is usually stateless and can be reused for all the work.

When serializing, you have to decide how to implement meta data serialization, which usually has to be the first thing in the stream as well as other features, such as link serialization. Look at `ReladomoSerializationContext`'s `serializeReladomoObject` and `serializeReladomoList` methods to understand the sequence of calls to the writer.

6.2. Deserialization

Your custom deserializer has to instantiate a `ReladomoDeserializer`. If the target class is known, you can construct the deserializer with it. If the class is in the stream, it has to be first and reading it should cause the `storeReladomoClassName` method to be called on the deserializer. When the deserialization is finished, you can get either a `Serialzied` or `SerializedList` object back, depending on what you deserialized, or rather if you called `startObject` or `startList` initially.