1. Goals

Our goal of this lesson is to introduce support for a new, binary format in the API, beyond just the standard XML and JSON.

2. Lesson Notes

First, we're going to add kryo support into the project:

```
<dependency>
    <groupId>com.esotericsoftware</groupId>
    <artifactId>kryo</artifactId>
    <version>4.0.0</version>
</dependency>
```

2.1. The new Kryo Http Message Converter

Now, we're going to make good use of the Spring HTTP message converters here.

That will enable support for Kryo representations:

```
public class KryoHttpMessageConverter extends AbstractHttpMessageConverter<Object> {
    public static final MediaType KRYO = new MediaType("application", "x-kryo");
    public KryoHttpMessageConverter() {
        super(KRYO);
    }
    @Override
    protected boolean supports(final Class<?> clazz) {
        return Object.class.isAssignableFrom(clazz);
```

```
    }
    @Override
    protected Object readInternal(final Class<? extends Object> clazz, final HttpInputMessage inputMessage) throws IOException {
        return null;
    }
    @Override
    protected void writeInternal(final Object object, final HttpOutputMessage outputMessage) throws IOException {
        //
    }
    @Override
    protected MediaType getDefaultContentType(final Object object) {
        return KRYO;
    }
}
```

We'll add the implementation later, for now let's wire it into the web configuration of the project - *UmWebConfig*.

First, we'll need to extend the more versatile *WebMvcConfigurationSupport*, and remove the *@EnableWebMvc*annotation.

And then let's add the Kryo message converter we just defined:

```
@Override
public void configureMessageConverters(final List<HttpMessageConverter<?>> messageConverters) {
    messageConverters.add(new KryoHttpMessageConverter());
    super.addDefaultHttpMessageConverters(messageConverters);
}
```

Now, let's finish the implementation of the message converter:

```
private final Kryo kryo = createKryo();
// ...
private static final Kryo createKryo() {
    final Kryo kryo = new Kryo();
    kryo.register(UserDto.class, 1);
    kryo.register(Role.class, 2);
```

```
    kryo.register(Privilege.class, 3);
    kryo.register(Principal.class, 4);
    return kryo;
}
```

Finally, here is the read implementation:

```
final Input input = new Input(inputMessage.getBody());
return kryo.readClassAndObject(input);
```

And the write:

```
final Output output = new Output(outputMessage.getBody());
kryo.writeClassAndObject(output, object);
output.flush();
```

But we’re not done.

We're using a single instance of Kryo here, which means that multiple threads will naturally be using this single instance.

However, what's critical to understand is that **the Kryo is not thread safe.**

And so we'll use a thread local here to have a Kryo instance per thread:

```
private static final ThreadLocal<Kryo> kryoThreadLocal = new ThreadLocal<Kryo>() {
    @Override
    protected Kryo initialValue() {
        return createKryo();
    }
};
```

Now we can simply access the instance out of this thread local:

```
kryoThreadLocal.get()
```

That's it - we're good to go.

2.2. The Live Test

Now that the implementation is done, with the system running - we can finally consume the API and ask for a Kryo representation of a Resource.

We're going to do that via a simple live test, but of course any HTTP Client is fine.

```
private static final String APPLICATION_KRYO = "application/x-kryo";
@Test
public void giveConsumingAsKryo_whenAllResourcesAreRetrieved_then200IsReceived() {
    // When
    Response response = getApi().givenReadAuthenticated().accept(APPLICATION_KRYO)
      .get(getApi().getUri());
    // Then
    assertThat(response.getStatusCode(), is(200));
    assertThat(response.getContentType(), equalTo(APPLICATION_KRYO));
}
```

3. Resources

- Binary Data Formats in a Spring REST API

- Http Message Converters with the Spring Framework