REST

Make Java REST development easier with the Jareto library

A small open source library helps with mapping Java exceptions and transporting HTTP metadata.

by Nenad Jovanovic

May 14, 2021

The Java ecosystem makes it easy to implement REST services. Using Java EE, Jakarta EE, JAX-RS, and JSON-B, the source code of a REST service looks like a plain Java program, enhanced with a few easy-to-understand annotations.

```
@Path("process")
@POST
public OutputBean process(InputBean input) {
  // process input, create & return output -
}
```

With the MicroProfile REST client, the same has become true for implementing the invoking side.

```
// create the MicroProfile REST client using
// can also be injected using @RestClient
IService service = RestClientBuilder.newBuild
  .baseUri(/* service URI, preferably from co
  .register(/* your preferred provider for JS
  .build(IService.class);
OutputBean output = service.process(input);
```

You don’t have to manually translate between the Java domain objects and their on-the-wire representation, since this is done automatically by JSON-B (as opposed to JSON-P).

However, there are several common use cases that still require additional boilerplate code if you want to remain on this level of abstraction. Jareto is a small Java library that provides useful

features in an easy-to-use way, for both server- and client-side development.

- For mapping Java exceptions: On the server side, Jareto helps with serialization to HTTP wire data (JSON); these JSON exceptions can also be parsed by non-Java clients and are customizable and allow transport of structured data. On the client side, Jareto helps with client deserialization from HTTP wire data (JSON).
- For transporting HTTP metadata: Jareto helps with HTTP status codes and headers.

Depending on your requirements, you can use the server-side part of Jareto, or the client-side part, or both.

Jareto, created by my company, SVC, is available on GitHub under the Apache License Version 2.0 License. There, you can also find a demo project with a sample web application and JUnit tests.

Jareto setup

To use the features provided by Jareto, add the following Maven dependencies to your project:

```
<!-- for using server-side features -->
<dependency>
  <groupId>at.co.svc.jareto</groupId>
  <artifactId>jareto-server</artifactId>
  <version>INSERT LATEST VERSION HERE</versio
</dependency>

<!-- for using client-side features -->
<dependency>
  <groupId>at.co.svc.jareto</groupId>
  <artifactId>jareto-client</artifactId>
  <version>INSERT LATEST VERSION HERE</versio
</dependency>
```

Jareto's features are implemented using JAX-RS providers. To prevent accidental activation of certain providers that you don't need or want (with future extensions in mind), Jareto politely abstains from autoregistration of providers. Instead, you must explicitly return the Jareto provider classes from the `getClasses` method of the server-side Application class, as follows:

```
public class BeanServiceConfig extends Applic

  @Override
  public Set<Class<?>> getClasses() {
    Set<Class<?>> classes = new HashSet<>();
    // add the Jareto providers for server-si
    classes.addAll(ServerProviders.getAll());
    // add your service's resource classes
    // ...
    return classes;
```

```
    }

}
```

Likewise, if you want to use Jareto in a REST client, you must register the Jareto provider classes during client construction.

```
RestClientBuilder builder = RestClientBuilder
ClientProviders.registerAll(builder);
```

Mapping Java exceptions

Without additional measures, throwing a runtime or checked exception from your service usually results in a stack trace being returned to the invoker. Apart from the security implications, this prevents clients from identifying and handling exceptional cases in an automated way, so this is usually not what you want.

JAX-RS specifies an unchecked WebApplicationException that allows you to customize both the HTTP response status and the returned payload. However, you would still have to

- Provide the payload using a lower-level JAX-RS response
- Take care of unexpected exceptions such as `NullPointerException` (by means of a global exception handler)

Jareto provides both a checked `AppException` and an unchecked `AppRuntimeException` that accept the following data during construction:

- Error message
- Error code
- Error detail code (optional)
- HTTP status code (optional; defaults to 500)

In the simplest possible case, an exception such as

```
throw new AppException("some-error-code", "so
```

will be automatically translated to the JSON HTTP response body as follows

```
{
    "code": "some-error-code",
    "text": "some error text"
}
```

Note that certain JSON-B implementations might also include the optional `detailCode` property with a `null` value.

Other types of `RuntimeException` are mapped using the following default values:

```
{
    "code": "UNEXPECTED_ERROR",
    "text": "An unexpected error has occurred
}
```

Jareto exposes a hook that is invoked during exception mapping, which lets you customize the default JSON attributes to be returned for `RuntimeException` and the logging behavior (no logging for Jareto exceptions; level ERROR for `RuntimeException`).

```
WebApplicationExceptionFactory.registerCustom
  // customize behavior by overriding the app
});
```

You can also return additional JSON attributes by extending `AppExceptionData` and passing it to the exception's constructor.

```
@JsonbPropertyOrder({ "code", "detailCode", "
public class CustomizedExceptionData extends

  // this bonus String shall also be transpor
  private String bonus;
  ...

}

...

throw new AppException(new CustomizedExceptio
```

In theory, transporting exception data would also be possible via HTTP headers (instead of JSON inside the HTTP response body). Even though this alternative might seem appealing at first glance, it does not scale to advanced use cases where exceptions will contain more-complex, structured data. In this respect, Jareto's exception handling resembles that of GraphQL, which is also capable of returning arbitrary error data inside the response payload.

By contrast, a MicroProfile REST client equipped with Jareto can catch and handle these exceptions, since they are automatically created from the incoming JSON representation.

```
try {
  service.process();
}
catch (AppException e) {
  System.out.println("error code: " + e.getDa
  System.out.println("error text: " + e.getDa
  System.out.println("HTTP response status: "
  System.out.println((CustomizedExceptionData
}
```

For security reasons (to prevent Java deserialization attacks), customized exception data types must be explicitly registered by the client, as follows:

```
ClientExceptionMapper.registerEntityClass(Cus
```

To summarize the exception mapping features, Jareto provides you with exception types that your service can throw and your client can catch, and that contain extensible, commonly used exception data—without the need for any additional boilerplate code. This is especially useful when creating new Java-based microservices that communicate with each other via REST. From the developer's perspective, throwing and catching exceptions feels the same as it is with plain, local invocations.

Transporting HTTP metadata

Although a MicroProfile REST client greatly simplifies invocations by (re-)using the service's Java interface, it does not offer any direct access to HTTP request headers or to HTTP response status and headers. To gain access to these, you would have to write additional `ResponseExceptionMapper` and `ClientRequestFilter` code. With Jareto, you can simply do the following on the client side:

```
// read HTTP response status
ClientResponse.CONTEXT.get().getStatus();

// read HTTP response header
ClientResponse.CONTEXT.get().getHeaderString(

// add HTTP request header
ClientRequestHeaders.addHeader("custom-header
```

The client-side API is deliberately designed to be usable also in standalone Java environments without Contexts and Dependency Injection (CDI). This way, running Java-based system tests against your REST service is quick and easy, as it does not require any CDI setup.

On the server side, simple access to the HTTP headers does not require anything other than `HttpServletRequest` and

`HttpServletResponse` from Jakarta EE.

```
@Context
private HttpServletRequest request;

@Context
private HttpServletResponse response;

public void process() {

  // read HTTP request header
  System.out.println(request.getHeader("reque

  // add HTTP response header
  response.addHeader("response-header-name",

}
```

Alternatively, reading HTTP request headers is also possible by using the JAX-RS annotation `HeaderParam`.

In case you would like to add a static HTTP header to your responses, you can do so by using Jareto's `@Header` annotation at the class or method level of your REST service interface or implementation.

```
@Path("service")
@Header(name = "class-response-header-name",
public interface IService {

  @Path("ping")
  @GET
  @Header(name = "method-response-header-name
  public String process();
```

Having already injected the `HttpServletResponse` for accessing the headers on the server, it might appear that setting the response status requires only one more line.

```
@Context
private HttpServletResponse response;

public void process() {

  // trying to set the HTTP status: does it w
  response.setStatus(201);

}
```

However, at least with WildFly 22 and Payara 5.2020.7, this line is not effective on its own. In addition, it requires an explicit flush, which lacks optical appeal and prevents later operations (such as response filters) from adding other headers.

```
@Context
private HttpServletResponse response;

public void process() {

  response.setStatus(201);
  try {
    response.flushBuffer();
  }
  catch (IOException e) {
    ...
  }

}
```

For this reason, Jareto allows you to set the HTTP status in the following way:

```
@Inject
private ServiceResponseBuilder responseBuilde

...

responseBuilder.get().status(DESIRED_STATUS);
```

Conclusion

When developing REST services and clients, transporting exceptions is a fundamental use case, but it still requires a certain amount of boilerplate code. Likewise, access to HTTP metadata (status code and headers) is not as convenient as it should be. Jareto addresses these issues by making just a few basic assumptions about how exceptions should be serialized to wire data and reducing the developer's part in the equation to the absolute minimum. In doing so, Jareto is

- Small (around 1,000 lines of code)
- Easy to understand and use

The features described in this article are based on the solid foundations provided by Jakarta EE and MicroProfile. On the shoulders of these giants, Jareto further improves the developer experience for creating new REST microservices.

Dig deeper

- Jareto documentation
- Jareto source code
- Jareto binaries

Nenad Jovanovic

Nenad Jovanovic is a Java developer, software architect, and enterprise architect at SVC, an Austrian company that develops e-health solutions in the public sector. SVC is the creator of the Jareto project. In his spare time, Jovanovic writes Java- and IT-related blog articles.

Share this Page