FREE eBook

LEARNING

cxf

Free unaffiliated eBook created from
Stack Overflow contributors.

#cxf

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cxf

It is an unofficial and free cxf ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cxf.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with cxf

Remarks

This section provides an overview of what cxf is, and why a developer might want to use it.

It should also mention any large subjects within cxf, and link out to the related topics. Since the Documentation for cxf is new, you may need to create initial versions of those related topics.

Examples

Basic Webclient with Provider

To get started we need a factory that produces WebClients.

```
public class ClientFactory {
    private Map<String, WebClient> cache = new HashMap<>();

    public enum RESTClient {
        PORTAL;
    }

    public WebClient fetchRestClient(RESTClient restClient) {

        if (this.cache.containsKey(restClient)) {
            return WebClient.fromClient(this.cache.get(rc));
        }

        if (RESTClient.enum.equals(rc)) {

            List<Object> providers = new ArrayList<Object>();
            providers.add(new GsonMessageBodyProvider());

            WebClient webClient = WebClient.create("https://blah.com", providers);

            HTTPConduit conduit = WebClient.getConfig(webClient).getHttpConduit();
            conduit.getClient().setReceiveTimeout(recieveTimeout);
            conduit.getClient().setConnectionTimeout(connectionTimout);

            this.cache.put(RESTClient.CAT_DEVELOPER_PORTAL.name(), webClient);
            return WebClient.fromClient(webClient);// thread safe
        }
    }
}
```

- First we create a list of providers (will get to those later)
- Next we make a new webclient using the static factory "create()". Here we can add a few other things like Basic Auth creds and thread safety. For now just use this one.
- Next we pull out the HTTPConduit and set the timeouts. CXF comes with Java base class clients but you can use Glassfish or others.
- Finally we cache the WebClient. This is important as the code so far is expensive to create.

The next line shows how to make it threadsafe. Notice this is also how we pull the code from the cache. Essentially we are making a model of the REST call and then cloning it each time we need it.

- Notice what is NOT here: We have not added any parameters or URLS. These can be added here but that would make a specific endpoint and we want a generic one. Also there are no headers added to the request. These do not make it past the "clone" so they must be added later.

Now we have a WebClient that is ready to go. Lets set up the rest of the call.

```
public Person fetchPerson(Long id) {
    long timer t = System.currentTimeMillis();
    Person person = null;
    try {
        wc = this.factory.findWebClient(RESTClient.PORTAL);
        wc.header(AUTH_HEADER, SUBSCRIPTION_KEY);
        wc.header(HttpHeaders.ACCEPT, "application/person-v1+json");

        wc.path("person").path("base");
        wc.query("id", String.valueOf(id));

        person = wc.get(Person.class);
    }
    catch (WebApplicationException wae) {
        // we wanna skip these. They will show up in the "finally" logs.
    }
    catch (Exception e) {
        log.error(MessageFormat.format("Error fetching Person: id:{0} ", id), e);
    }
    finally {
        log.info("GET HTTP:{} - Time:[{}ms] - URL:{} - Content-Type:{}",
wc.getResponse().getStatus(), (System.currentTimeMillis() - timer), wc.getCurrentURI(),
wc.getResponse().getMetadata().get("content-type"));
        wc.close();
    }
    return p;
}
```

- Inside the "try" we grab a WebClient from the factory. This is a new "frosty" one.
- Next we set some headers. Here we add some kind of Auth header and then an accept header. Notice we have a custom accept header.
- Adding the path and query strings comes next. Keep in mind there is no order to these steps.
- Finally we do the "get". There are several ways to do this of course. Here we have CXF do the JSON mapping for us. When done this way we have to deal with the WebApplicationExceptions. So if we get a 404, CXF will throw an exception. Notice here I eat those exceptions because I just log the response in the finally. I do however want to get any OTHER exception as they might be important.
- If you don't like this Exception handling switching you can get the Response object back from the "get". This object holds the entity and the HTTP status code. It will NEVER throw a WebApplicationException. The only drawback is it does not do the JSON mapping for you.
- Finally, in the "finally" clause we have a "wc.close()". If you get the object from the get Clause you don't really have to do this. Something might go wrong though so it is a good failsafe.

So, what about that "accept" header: application/person-v1+json How will CXF know how to parse it? CXF comes with some built in JSON parsers but I wanted to use Gson. Many of the other implementations of Json parsers need some kind of annotation but not Gson. It is stupid easy to use.

```
public class GsonMessageBodyProvider<T> implements MessageBodyReader<T>, MessageBodyWriter<T>
{

    private Gson gson = new GsonBuilder().create();

    @Override
    public boolean isReadable(Class<?> type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
        return StringUtils.endsWithIgnoreCase(mediaType.getSubtype(), "json");
    }

    @Override
    public T readFrom(Class<T> type, Type genericType, Annotation[] annotations, MediaType
mediaType, MultivaluedMap<String, String> httpHeaders, InputStream entityStream) throws
IOException, WebApplicationException {
        try {
            return gson.fromJson(new BufferedReader(new InputStreamReader(entityStream, "UTF-
8")), type);
        }
        catch (Exception e) {
            throw new IOException("Trouble reading into:" + type.getName(), e);
        }
    }

    @Override
    public boolean isWriteable(Class<?> type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
        return StringUtils.containsIgnoreCase(mediaType.getSubtype(), "json");
    }

    @Override
    public long getSize(T t, Class<?> type, Type genericType, Annotation[] annotations,
MediaType mediaType) {
        return 0;
    }

    @Override
    public void writeTo(T t, Class<?> type, Type genericType, Annotation[] annotations,
MediaType mediaType, MultivaluedMap<String, Object> httpHeaders, OutputStream entityStream)
throws IOException, WebApplicationException {
        try {
            JsonWriter writer = new JsonWriter(new OutputStreamWriter(entityStream, "UTF-8"));
            writer.setIndent("  ");
            gson.toJson(t, type, writer);
            writer.close();
        }
        catch (Exception e) {
            throw new IOException("Trouble marshalling:" + type.getName(), e);
        }
    }
}
```

The code is straight forward. You implement two interfaces: MessageBodyReader and MessageBodyWriter (or just one) and then add it to the "providers" when creating the WebClient.

CXF figures it out from there. One option is to return "true" in the "isReadable()" "isWriteable()" methods. This will assure that CXF uses this class instead of all the built in ones. The added providers will be checked first.

Setting up CXF for JAX-RS

The jars for CXF JAX-RS are found in Maven:

```
<!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-rs-client -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-client</artifactId>
    <version>3.1.10</version>
</dependency>
```

These jars are all you need to get it running:

```
cxf-rt-rs-client-3.1.10.jar
cxf-rt-transports-http-3.1.10.jar
cxf-core-3.1.10.jar
woodstox-core-asl-4.4.1.jar
stax2-api-3.1.4.jar
xmlschema-core-2.2.1.jar
cxf-rt-frontend-jaxrs-3.1.10.jar
javax.ws.rs-api-2.0.1.jar
javax.annotation-api-1.2.jar
```

Client Filters

One good reason to use Filters is for logging. Using this technique a REST call can be logged and timed easily.

```
public class RestLogger implements ClientRequestFilter, ClientResponseFilter {
    private static final Logger log = LoggerFactory.getLogger(RestLogger.class);

    // Used for timing this call.
    private static final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private boolean logRequestEntity;
    private boolean logResponseEntity;

    private static Gson GSON = new GsonBuilder().create();

    public RestLogger(boolean logRequestEntity, boolean logResponseEntity) {
        this.logRequestEntity = logRequestEntity;
        this.logResponseEntity = logResponseEntity;
    }


    @Override
    public void filter(ClientRequestContext requestContext) throws IOException {
        startTime.set(System.currentTimeMillis());
    }

    @Override
    public void filter(ClientRequestContext requestContext, ClientResponseContext
```

```
responseContext) throws IOException {
        StringBuilder sb = new StringBuilder();
        sb.append("HTTP:").append(responseContext.getStatus());
        sb.append(" - Time:[").append(System.currentTimeMillis() -
startTime.get().longValue()).append("ms]");
        sb.append(" - Path:").append(requestContext.getUri());
        sb.append(" - Content-
type:").append(requestContext.getStringHeaders().getFirst(HttpHeaders.CONTENT_TYPE.toString()));

        sb.append(" -
Accept:").append(requestContext.getStringHeaders().getFirst(HttpHeaders.ACCEPT.toString()));
        if (logRequestEntity) {
            sb.append(" - RequestBody:").append(requestContext.getEntity() != null ?
GSON.toJson(requestContext.getEntity()) : "none");
        }
        if (logResponseEntity) {
            sb.append(" - ResponseBody:").append(this.logResponse(responseContext));
        }
        log.info(sb.toString());
    }

    private String logResponse(ClientResponseContext response) {
        StringBuilder b = new StringBuilder();
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        InputStream in = response.getEntityStream();
        try {
            ReaderWriter.writeTo(in, out);
            byte[] requestEntity = out.toByteArray();
            b.append(new String(requestEntity));
            response.setEntityStream(new ByteArrayInputStream(requestEntity));
        }
        catch (IOException ex) {
            throw new ClientHandlerException(ex);
        }
        return b.toString();
    }
}
```

Above you can see the request is intercepted before the response is sent and a ThreadLocal Long is set. When the response is returned we can log the request and response and all kinds of pertinent data. Of course this only works for Gson responses and such but can be modified easily. This is set up this way:

```
List<Object> providers = new ArrayList<Object>();
providers.add(new GsonMessageBodyProvider());
providers.add(new RestLogger(true, true)); <------right here!

WebClient webClient = WebClient.create(PORTAL_URL, providers);
```

The log provided should look something like this:

```
7278 [main] INFO  blah.RestLogger - HTTP:200 - Time:[1391ms] - User:unknown -
Path:https://blah.com/tmet/moduleDescriptions/desc?languageCode=en&moduleId=142 - Content-
type:null - Accept:application/json - RequestBody:none -
ResponseBody:{"languageCode":"EN","moduleId":142,"moduleDescription":"ECAP"}
```

Read Getting started with cxf online: https://riptutorial.com/cxf/topic/7387/getting-started-with-cxf

Credits

S. No	Chapters	Contributors
1	Getting started with cxf	Community, markthegrea