JAX-RS

JAX-RS

Introduction

JAX-RS: Java API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural style.

CXF supports JAX-RS 2.1 (JSR-370), 2.0 (JSR-339) and 1.1 (JSR-311).

CXF 3.2.0 supports JAX-RS 2.1. All existing JAX-RS 2.0 and 1.1 applications can be run with CXF 3.2.0.

CXF 3.1.x and 3.0.x support JAX-RS 2.0. Existing JAX-RS 1.1 applications can be run with CXF 3.1.x/3.0.x.

See below for more information about the compliance.

JAX-RS related demos are located under the samples/jax_rs directory.

Outstanding JAX-RS JIRA issues can be found here.

JAX-RS Compliance

2.1 Final

CXF 3.2.0 has been updated to implement the JAX-RS 2.1 API's as completely as possible.

If another TCK licensee that uses CXF's JAX-RS 2.1 implementation in their products finds issues with CXF's compliance, we are more than happy to fix bugs that are raised.

2.0 Final

CXF 3.1.x and CXF 3.0.x have been updated to implement the JAX-RS 2.0 API's as completely as possible without access to the final JAX-RS 2.0 TCK.
We have done extensive testing with JAX-RS 2.0 user applications, samples, and the preliminary TCK to make sure CXF's implementation is as complete and compatible as we can make it.
CXF makes and will continue making the best possible effort to have JAX-RS 2.0 and new JAX-RS version implementations technically complete and offering an environment for running the portable JAX-RS 2.0 applications.
If the final 2.0 TCK is made available to Apache, we will make sure CXF is updated to pass.
If another TCK licensee that uses CXF's JAX-RS 2.0 implementation in their products finds issues with CXF's compliance, we are more than happy to fix bugs that are raised.

1.1

Apache CXF 2.6.x passes the final JAX-RS 1.1 TCK and is formally 1.1 compliant.

Please consult the TomEE documentation on the support of Java EE related JAX-RS 1.1 options in its Apache CXF-based JAX-RS runtime.

CXF 2.7.x and CXF 3.0.0 will fully support and run JAX-RS 1.1 applications but will not pass the JAX-RS 1.1 TCK Signature tests due to

CXF 2.7.x and CXF 3.0.0 depending on 2.0-m10 and 2.0 final versions of JAX-RS 2.0 API.

Project setup and configuration

Migration

From JAX-RS 2.0 to JAX-RS 2.1

JAX-RS 2.1 is backward compatible with JAX-RS 2.0. Please see JAX-RS Basics for more information about JAX-RS 2.1.

All the existing JAX-RS 2.0 and 1.1 applications will run on CXF 3.2.0.

From JAX-RS 1.1 to 2.0

JAX-RS 2.0 is backward compatible with JAX-RS 1.1. Please see JAX-RS Basics for more information about JAX-RS 2.0.

CXF 3.1.x and CXF 3.0.x are expected to support the existing JAX-RS 1.1 applications.

From CXF 2.7.x to CXF 3.0.x or 3.1.x

Please check the CXF 3.0.0 Migration Guide for the information about all the changes
in CXF 3.0.0. Here are more details on the changes specifically affecting JAX-RS users:

1. CXF RequestHandler and ResponseHandler filters have been removed.

These legacy CXF filters are still supported in 2.7.x but no longer in 3.0.0. Please use ContainerRequestFilter and ContainerResponseFilter instead. Also, ReaderInterceptor and WriterInterceptor can be used too.

Note, CXF filters had org.apache.cxf.message.Message available in the signature. If CXF Message is used in the existing CXF RequestHandler or ResponseHandler then use "org.apache.cxf.phase.PhaseInterceptorChain.getCurrentMessage()" or "org.apache.cxf.jaxrs.util.JAXRSUtils.getCurrentMessage()" to get a Message which has all the contextual information available.

For example, instead of

```
public class CustomRequestHandler implements RequestHandler {
    public Response handleRequest(Message message, ClassResourceInfo cri) {
    }
}

public class CustomResponseHandler implements ResponseHandler {
    public Response handleResponse(Message message, OperationResourceInfo
ori, Response response) {
    }
}
```

do

```
public class CustomRequestFilter implements ContainerRequestFilter {
    public void filter(ContainerRequestContext context) {
        Message message = JAXRSUtils.getCurrentMessage();
       ClassResourceInfo cri = message.getExchange().get
(OperationResourceInfo.class).getClassResourceInfo();

        // or consider using JAX-RS 2.0 ResourceInfo context

        // finally use context.abortWith(Response) if you need to block
the request
    }
}

public class CustomResponseFilter implements ContainerResponseFilter {
    public void filter(ContainerRequestContext inContext,
ContainerResponseContext outContext) {
        Message message = JAXRSUtils.getCurrentMessage();
        OperationResourceInfo cri = message.getExchange().get
(OperationResourceInfo.class);

        // or consider using JAX-RS 2.0 ResourceInfo context

        // finally, work with ContainerResponseContext to modify specific
Response properties
    }
}
```

The above is only needed to ease the migration of the existing RequestHandler or ResponseHandler implementations. Prefer writing portable JAX-RS 2.0 filter implementations if possible. CXF interceptors can be used to do the CXF specific code if needed.

2. CXF org.apache.cxf.jaxrs.ext.form.Form has been dropped, please use JAX-RS 2.0 Form instead. For example, use:

```
import javax.ws.rs.core.Form;

Form form = new Form().param("a", "b");
```

instead of

```
import org.apache.cxf.jaxrs.ext.form.Form;

Form form = new Form().set("a", "b");
```

3. CXF WebClient and proxy code has been moved to a new cxf-rt-rs-client module.
Also, jaxrs:client elements for injecting proxies have had the namespace changed from from "http://cxf.apache.org/jaxrs" to "http://cxf.apache.org/jaxrs-client".

Please see JAX-RS Client API page for more information.

4. WADL Auto Generator code has been moved to a new cxf-rt-rs-service-description module.

5. CXF ParameterHandler has been dropped. Please use ParameterConverterProvider instead, it can be used both on the server and client sides.

6. JAX-RS 2.0 introduces a controversial requirement that the default built-in JAX-RS MessageBodyWriter and JAX-RS MessageBodyReader providers are **preferred** to custom providers supporting the same types unless the custom providers are precisely typed, for example, if you have a custom InputStream reader properly implementing isReadable:

```
public class MyStreamProvider implements MessageBodyReader<Object> {
    public boolean isReadable(Class<?> cls, ...) {
        return InputStream.class.isAssignableFrom(cls) || Reader.class.
isAssignableFrom(cls);
    }
    // other methods
}
```

then the runtime will ignore it and choose a default InputStream/Reader reader because MyStreamProvider is typed on Object. This was done to deal with the cases where well-known JSON/etc providers are blindly supporting all types in their isReadable methods by always returning 'true' and then failing when asked to actually read the incoming stream into InputStream/etc directly. In case of MyStreamProvider, it will need to be split into MyInputStreamProvider and MyReaderProvider typed on InputStream and Reader respectively.

At CXF level, the users which depend on CXF MultipartProvider to have InputStream or String references to multipart attachments will be affected unless they use @Multipart annotation. For example, if we have a multipart payload with a single part/attachment only then the following code:

```
@POST
@Consumes("multipart/form-data")
public void upload(InputStream is) {
}
```

which in CXF 2.7.x or earlier will return a pointer to first/single individual part, will actually return a stream representing the complete unprocessed multipart payload. Adding a @Multipart marker will keep the existing code working as expected:

```
@POST
@Consumes("multipart/form-data")
public void upload(@Multipart InputStream is) {
}
```

Alternatively, setting a "support.type.as.multipart" contextual property will do.

7. If the custom code throws JAX-RS WebApplicationException with Response containing a non-null entity then custom WebApplicationException mappers will be bypassed - another problematic requirement, for example, the custom mappers doing the logging will miss on such exceptions. Set CXF "support.wae.spec.optimization" property to false to disable it.

8. In some cases the matching sub-resource locators will be dropped to precisely meet the current JAX-RS matching algorithm text, please see CXF-5650 for more information. Use a new "keep.subresource.candidates" property to support the existing application if needed.

CXF 3.1.2 Provider Sorting Changes

Starting from CXF 3.1.2 customMessageBodyReader (MBR), MessageBodyWriter (MBW) and ExceptionMapper providers are sorted together with default providers.

Before CXF 3.1.2 if a custom MBR or MBW matches the read or write selection criteria, example, if MBR Consumes matches Content-Type and its isReadable() returns true, then

the default providers are not even checked. The specification however does let the custom providers be selected only if no higher priority matching default provider is available.

For example, suppose you have a custom StringReader which is not typed by String but by Object. In this case the default provider which is typed by String wins. To have the custom String provider winning one needs to type it by String.

Check the specification or ask at the users list for more details.

Maven dependencies

CXF 3.2.0

The cxf-rt-frontend-jaxrs dependency is required:

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-frontend-jaxrs</artifactId>
   <version>3.2.0</version>
</dependency>
```

This will in turn pull other CXF modules such `cxf-core` and `cxf-rt-transports-http`, check the pom for more information.

`javax.ws.rs/javax.ws.rs-api/2.1` dependency provides JAX-RS 2.1 Final API.

CXF 3.1.x

The cxf-rt-frontend-jaxrs dependency is required:

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-frontend-jaxrs</artifactId>
   <version>3.1.12</version>
</dependency>
```

This will in turn pull other CXF modules such `cxf-core` and `cxf-rt-transports-http`, check the pom for more information.

`javax.ws.rs/javax.ws.rs-api/2.0` dependency provides JAX-RS 2.0 Final API.

`javax.annotation/javax.annotation-api/1.2 dependency is needed if custom JAX-RS 2.0 filters or interceptors use a javax.annotation.Priority annotation.`

Existing JAX-RS 1.1 applications can run in CXF 3.1.x and CXF 3.0.x.

CXF JAX-RS bundle

Note CXF JAX-RS bundle has been removed in CXF 3.0.0. Prefer depending on the JAX-RS frontend directly. In CXF 3.0.0 a complete CXF all-inclusive bundle can still be used if really needed.

Only in CXF 2.7.x or earlier:
A standalone JAX-RS bundle is available which may be of interest to users doing the JAX-RS work only.

Please note that this bundle has a transitive Maven dependency on the Jetty server modules. If you are using Maven and working with other servlet containers such as Tomcat then please add the following exclusion:

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-bundle-jaxrs</artifactId>
   <version>${cxf.version}</version>
   <exclusions>
       <exclusion>
         <groupId>org.eclipse.jetty</groupId>
         <artifactId>jetty-server</artifactId>
       </exclusion>
   </exclusions>

</dependency>
```

What is New

- Complete support for JAX-RS 2.1, please see JAX-RS Basics for more information
- JAX-RS NIO extension based on the early JAX-RS 2.1 API prototype.
- JAX-RS RxJava Observable support: as a standard JAX-RS 2.1 RxInvoker client provider and returning it asynchronously from the resource methods (CXF extension)
- JAX-RS Project Reactor Support also based on the JAX-RS 2.1 RxInvoker paradigm
- Complete support for JAX-RS 2.0, please see JAX-RS Basics for more information
- Bean Validation 1.1 Support, please see http://cxf.apache.org/docs/validationfeature.html for more information
- Swagger Feature for generating Swagger API documentation from JAX-RS endpoints

Getting Started with JAX-RS

Understanding the Basics

You are encouraged to read JAX-RS 2.1 JSR-370 specification to find out the information not covered by this documentation. The specification enhances JAX-RS 2.0 by introducing a support for Reactive Client API extensions, Server Sent Events (client and server), returning CompletableFuture from the resource methods and the sub-resource classes (as opposed to instances) from the sub-resource locators.

You are also encouraged to read JAX-RS 2.0 JSR-339 specification to find out the information not covered by this documentation. The specification introduces many terms such as root resources, resource methods, sub-resources and sub-resource locators, message body readers and writers. JAX-RS 2.0 additionally introduces filters, interceptors, new client API, features, new exception classes, server-side support for asynchronous invocations.

Please see the JAX-RS Basics page for more information.

Support for Data Bindings

JAX-RS MessageBodyReader and MessageBodyWriter can be used to create data bindings for reading and writing data in a number of different formats. Compliant JAX-RS implementations are expected to support JAXB-annotated beans, JAXP Source objects, InputStreams, etc.

In addition, CXF JAX-RS lets users reuse existing CXF DataBindings for working with JAXB, XBeans, Aegis and SDO.

Please see the JAX-RS Data Bindings page for more information.

How Request URI is Matched

Lets assume you have a web application called 'rest' (example, a 'rest.war' archive). CXFServlet's url-pattern is "/test/*". Finally, jaxrs:server's address is "/bar".

Requests like /rest/test/bar or /rest/test/bar/baz will be delivered to one of the resource classes in a given jaxrs:server endpoint. For the former request to be handled, a resource class with @Path("/") should be available, in the latter case - at least @Path("/") or a more specific @Path("/baz").

The same requirement can be expressed by having a CXFServlet with "/*" and jaxrs:server with "/test/bar".

When both CXFServlet and jaxrs:server use "/" then it's a root resource class which should provide a @Path with at least "/test/bar" for the above requests to be matched.

Generally, it can be a good idea to specify the URI segments which are more likely to change now and then with CXFServlets or jaxrs:server.

Client API

CXF 3.0.0 implements JAX-RS 2.0 Client API.

CXF 2.7.x or earlier provides a comprehensive support for developing RESTful clients by supporting 3 flavors of the client API: proxy-based, HTTP-centric and XML-centric. CXF-specific client API is supported alongside new JAX-RS 2.0 Client API in CXF 3.0.0.

Please see the JAX-RS Client API page for more information.

Bean Validation

Bean Validation 1.1 is supported since CXF 3.0.0-milestone1. Please see the http://cxf.apache.org/docs/validationfeature.html for more information.

Filters, Interceptors and Invokers

It is possible to intercept and modify the inbound and outbound calls with the help of CXF JAX-RS filters and/or CXF interceptors. Additionally, custom invokers offer an option to intercept a call immediately before a service bean is invoked.

Please see the JAX-RS Filters page for more information.

Please see the JAX-RS Basics page for more information about new JAX-RS 2.0 filters and interceptors available in CXF 2.7.x and 3.0.0.

Service listings and WADL support

New: Swagger feature has been introduced.

CXF JAX-RS supports WADL. CXF JAX-RS service endpoints can be listed in the service listings page and users can check the WADL documents.

Please see the JAXRS Services Description page for more information.

Configuring JAX-RS services

JAX-RS services can be configured programmatically, using Blueprint, Spring or CXFNonSpringJAXRSServlet.

Please see the JAXRS Services Configuration page for more information.

Testing

JAX-RS services can be easily tested using the embedded Jetty or CXF Local Transport.
Please see the JAXRS Testing page for more information.

Debugging

One may want to use a browser to test how a given HTTP resource reacts to different HTTP Accept or Accept-Language header values and request methods. For example, if a resource class supports a "/resource" URI then one can test the resource class using one of the following queries :

```
> GET /resource.xml
> GET /resource.en
```

The runtime will replace '.xml' or '.en' with an appropriate header value. For it to know the type or language value associated with a given URI suffix, some configuration needs to be done. Here's an example of how it can be done with Spring:

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:extensionMappings>
    <entry key="json" value="application/json"/>
    <entry key="xml" value="application/xml"/>
  </jaxrs:extensionMappings>
  <jaxrs:languageMappings>
     <entry key="en" value="en-gb"/>
  </jaxrs:languageMappings>
</jaxrs:server>
```

CXF also supports a _type query as an alternative to appending extensions like '.xml' to request URIs:

{{ > GET /resource?_type=xml}}

Overriding a request method is also easy:

```
> GET /resource?_method=POST
```

Alternatively, one can specify an HTTP header X-HTTP-Method-Override:

```
> POST /books
> X-HTTP-Method-Override : PATCH
```

For example, at the moment the http-centric client API does not support arbitrary HTTP verbs except for those supported by Java HTTPUrlConnection. When needed, X-HTTP-Method-Override can be set to overcome this limitation.

Finally, a "_ctype" query allows for overriding Content-Type.

Please see the Debugging and Logging page for more information on how to debug and log service calls in CXF.

Logging

Many of the existing CXF features can be applied either to `jaxrs:server` or `jaxrs:client`. For example, to enable logging of requests and responses, simply do:

```
<beans xmlns:cxf="http://cxf.apache.org/core"
   xsi:schemaLocation="http://cxf.apache.org/core
      http://cxf.apache.org/schemas/core.xsd">
<jaxrs:server>
<jaxrs:features>
     <cxf:logging/>
</jaxrs:features>
<jaxrs:server>
</beans>
```

Please make sure the `http://cxf.apache.org/core` namespace is in scope.

Starting from CXF 2.3.0 it is also possible to convert log events into Atom entries and either push them to receivers or make them available for polling.

Please see the Debugging and Logging page for more information.

Advanced Features

Multiparts

Multiparts can be handled in a number of ways. The CXF core runtime provides advanced support for handling attachments which CXF JAX-RS builds upon.

Please see the JAX-RS Multiparts page for more information.

Secure JAX-RS services

Transport level HTTPS security can be used to protect messages exchanged between CXF JAX-RS endpoints and providers.

Authentication and authorization can be enforced in a number of ways.

Please see the Secure JAX-RS Services page for more information.

Please also check JAX-RS XML Security, JAX-RS SAML and JAX-RS OAuth2 pages for more information about the advanced security topics.

Failover and Load Distribution Features

Starting from CXF 2.4.1, CXF JAX-RS proxy and WebClient consumers can be backed up by failover and load distribution features.
Please see the JAX-RS Failover page for more information.

Redirection

Starting from CXF 2.2.5 it is possible to redirect the request or response call to other servlet resources by configuring CXFServlet or using CXF JAX-RS RequestDispatcherProvider.

Please see the JAX-RS Redirection page for more information.

XSLT and XPath

XSLT and XPath are promoted and treated as first-class citizens in CXF JAX-RS. These technologies can be very powerful when generating complex data or retrieving data of interest out of complex XML fragments.

Please see the JAX-RS Advanced XML page for more information.

Complex Search Queries

Using query parameter beans provides a way to capture search requirements that can be expressed by enumerating name/value pairs, for example, a query such as '?name=CXF&version=2.3' can be captured by a bean containing setName and setVersion methods. This 'template' bean can be used in the code to compare it against all available local data.

Versions 2.3 and later of CXF JAXRS support another option for doing advanced search queries using the Feed Item Query Language(FIQL).

Please see the JAX-RS Search page for more information.

Model-View-Controller support

XSLT
Please see the JAX-RS Advanced XML page for more information. on how `XSLTJaxbProvider` can be used to generate complex (X)HTML views.

JSP

With the introduction of `RequestDispatcherProvider` it is now possible for JAXRS service responses be redirected to JSP pages for further processing. Please see the JAX-RS Redirection page for more information.

Combining JAX-WS and JAX-RS

CXF JAX-RS tries to make it easy for SOAP developers to experiment with JAX-RS and combine both JAX-WS and JAX-RS in the same service bean when needed.

Please see the JAX-RS and JAX-WS page for more information.

Integration with Distributed OSGi

Distributed OSGi RI is a CXF subproject. DOSGi mandates how registered Java interfaces can be exposed
and consumed as remote services. DOSGi single and multi bundle distributions contain all the OSGI bundles required for a CXF endpoint be successfully published.

CXF JAX-RS implementations has been integrated with DOSGi RI 1.1-SNAPSHOT which makes it possible to expose Java interfaces as RESTful services and consume such services using a proxy-based client API.

Please see the DOSGI Reference page ('org.apache.cxf.rs' properties) and a greeter_rest sample for more information. Note that this demo can be run exactly as a SOAP-based greeter demo as it registers and consumes a similar (but) JAX-RS annotated GreeterService. In addition, this demo shows how one can register and consume a given interface (GreeterService2) without using explicit JAX-RS annotations but providing an out-of-band user model description.

OData Support

CXF JAX-RS endpoints can support OData in two ways by relying on Apache Olingo.

First, the OData "$filter" query is supported by the Search extension where an endpoint with the application specific API can respond to the filter queries, for example, return a collection of books matching the fillter search criteria.

Second, CXF JAX-RS can be used to interpose over the Olingo, as is demoed here. Effectively such a CXF endpoint becomes an OData server: all it does it delegates to Olingo. The idea is to be able to add CXF specific features and interceptors in front of Olingo.

Other Advanced Features

CXF JAX-RS provides a number of advanced extensions such as the support for the JMS transport, one-way invocations (HTTP and JMS), suspended invocations (HTTP and JMS), making existing code REST-aware by applying external user models, etc.

Please see the JAX-RS Advanced Features page for more information.

Maven Plugins

Please see the JAX-RS Maven Plugins page for more information about the Maven plugins and archetypes which can help with creating CXF JAX-RS applications.

Deployment

CXF JAX-RS applications packaged as WAR archives can be deployed into standalone Servlet containers such as Tomcat or Jetty.
Please see the JAX-RS Deployment page for the tips on how to deploy the CXF JAX-RS applications into various Java EE and OSGI application servers successfully.

Third-party projects

- REST Utilities: RESTUtils

References

- JSR-000311 JAX-RS: The JavaTM API for RESTful Web Services
- Architectural Styles and the Design of Network-based Software Architectures
- Representational State Transfer - Wikipedia
- RESTful Web Services Cookbook - Solutions for Improving Scalability and Simplicity *by Subbu Allamarajuy* (O'Reilly Media, February 2010)
- RESTful Java with JAX-RS *by Bill Burke* (O'Reilly Media, November 2009)
- Java Web Services: Up and Running *by Martin Kalin* (O'Reilly Media, February 2009)
- RESTful Web Services - Web services for the real world *by Leonard Richardson, Sam Ruby* (O'Reilly Media, May 2007)
- RESTful Web Services *by Sameer Tyagi* (Oracle , August 2006)
- RESTful Web Services - "Unofficial homepage for a book about simple web services." *Unknown*
- How I Explained REST to My Wife *by Ryan Tomayko* (http://tomayko.com, December 2004)

How to contribute

CXF JAX-RS implementation sits on top of the core CXF runtime and is quite self-contained and isolated from other CXF modules such as jaxws and simple frontends.

Please check the issue list and see if you are interested in fixing one of the issues.

If you decide to go ahead then the fastest way to start is to

- do the fast trunk build using '`mvn install -Pfastinstall`'
- setup the workspace 'mvn -Psetup.eclipse' which will create a workspace in a 'workspace' folder, next to 'trunk'
- import cxf modules from the trunk into the workspace and start working with the cxf-frontend-jaxrs module

If you are about to submit a patch after building a trunk/rt/frontend/jaxrs, then please also run JAX-RS system tests in trunk/systests/jaxrs :
`> mvn install`

You can also check out the general Getting Involved web page for more information on contributing.