Annotations

CXF provides several custom annotations that can be used to configure and customize the CXF runtime.

org.apache.cxf.feature.Features

The @Features annotation is used to add Features. See the FeaturesList for the list of Features we provide "out of the box", but you can easily create your own. In many cases, however, those features have Annotations themselves which can be used and provide greater control over configuration.

org.apache.cxf.interceptor.InInterceptors, org.apache.cxf.interceptor.OutInterceptors, org.apache.cxf.interceptor.OutFaultInterceptors, org.apache.cxf.interceptor.InFaultInterceptors

Add interceptors to the various chains used to process messages. See Interceptors for more detail.

org.apache.cxf.annotations.WSDLDocumentation org.apache.cxf.annotations.WSDLDocumentationCollection (since 2.3)

For "java first" scenarios where the WSDL is derived from the Java interfaces/code, these annotations allow adding wsd:documentation elements to various locations in the generated wsdl.

For example:

```
@WebService
@WSDLDocumentationCollection(
    {
        @WSDLDocumentation("My portType documentation"),
        @WSDLDocumentation(value = "My top level documentation",
                           placement = WSDLDocumentation.Placement.TOP),
        @WSDLDocumentation(value = "My binding doc",
                           placement = WSDLDocumentation.Placement.BINDING)
    }
)
public interface MyService {

    @WSDLDocumentation("The docs for echoString")
    String echoString(String s);

}
```

org.apache.cxf.annotations.SchemaValidation (since 2.3)

Turns on SchemaValidation for messages. By default, for performance reasons, CXF does not validate message against the schema. By turning on validation, problems with messages not matching the schema are easier to determine. These annotations can be applied globally to the SEI, but also can be applied at method level in both the SEI and Implementation.

IN	Apply schema validation to INcoming messages on client and server
REQUEST	**(Since 2.7.14, 3.0.3, 3.1**) Apply schema validation to Request messages. This annotation will cause validation to be applied to OUTgoing Client messages, and INcoming Server messages.
OUT	Apply schema validation to OUTgoing messages on Client and Server
RESPONSE	**(Since 2.7.14, 3.0.3, 3.1)** Apply schema validation to Response messages. This annotation will cause validation to be applied to INcoming Client messages, and OUTgoing Server messages.
BOTH	Apply schema validation to both INcoming and OUTgoing messages on Client and Server
NONE	All schema validation is disabled.

```
@WebService
@SchemaValidation(type = SchemaValidationType.BOTH)
public interface MyService {
    Foo validateBoth(Bar data);

    @SchemaValidation(type = SchemaValidationType.NONE)
    Foo validateNone(Bar data);

    @SchemaValidation(type = SchemaValidationType.IN)
    Foo validateIn(Bar data);

    @SchemaValidation(type = SchemaValidationType.OUT)
    Foo validateOut(Bar data);

    @SchemaValidation(type = SchemaValidationType.REQUEST)
    Foo validateRequest(Bar data);

    @SchemaValidation(type = SchemaValidationType.RESPONSE)
    Foo validateResponse(Bar data);
}
```

org.apache.cxf.annotations.DataBinding (since 2.2.4)

Sets the DataBinding class that is associated with the service. By default, CXF assumes you are using the JAXB data binding. However, CXF supports different databindings such as XMLBeans, Aegis, SDO, and possibly more. This annotation can be used in place of configuration to select the databinding class.

```
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface MyService {
    public commonj.sdo.DataObject echoStruct(
        commonj.sdo.DataObject struct
    );
}
```

org.apache.cxf.annotations.Logging (since 2.3)

Turns on logging for the endpoint. Can be used to control the size limits of what gets logged as well as the location. It supports the following attributes:

limit	Sets the size limit after which the message is truncated in the logs. Default is 64K
inLocation	Sets the location to log incoming messages. Can be <stderr>, <stdout>, <logger>, or a file: URL. Default is <logger>
outLocation	Sets the location to log outgoing messages. Can be <stderr>, <stdout>, <logger>, or a file: URL. Default is <logger>

```
@Logging(limit=16000, inLocation="<stdout>")
public interface MyService {

    String echoString(String s);

}
```

org.apache.cxf.annotations.GZIP (since 2.3)

Enables GZIP compression of on-the-wire data. Supported attributes:

threshold	the threshold under which messages are not gzipped
force	force GZIP compression instead of negotiating via the Accept-Encoding header

GZIP is a negotiated enhancement. An initial request from a client will not be gzipped, but an Accept header will be added and if the server supports it, the response will be gzipped and any subsequent requests will be.

org.apache.cxf.annotations.FastInfoset (since 2.3)

Enables FastInfoset of on-the-wire data. Supported attributes:

force	forces the use of fastinfoset instead of negotiating. Default is false

FastInfoset is a negotiated enhancement. An initial request from a client will not be in fastinfoset, but an Accept header will be added and if the server supports it, the response will be in fastinfoset and any subsequent requests will be.

org.apache.cxf.annotations.EndpointProperty
org.apache.cxf.annotations.EndpointProperties (since 2.3)

Adds a property to an endpoint. Many things such as WS-Security related things and such can be configured via endpoint properties. Traditionally, these would be set via the <jaxws:properties> element on the <jaxws:endpoint> element in the spring config, but these annotations allow these properties to be configured into the code.

```
@WebService
@EndpointProperties(
    {
       @EndpointProperty(key = "my.property", value="some value"),
       @EndpointProperty(key = "my.other.property", value="some other value"),
    })
public interface MyService {
    String echoString(String s);
}
```

org.apache.cxf.annotations.Policy
org.apache.cxf.annotations.Policies (since 2.3)

Used to attach WS-Policy fragments to a service or operation. The Policy supports the attributes:

uri	**REQUIRED** the location of the file containing the Policy definition
includeInWSDL	Whether to include the policy in the generated WSDL when generating a wsdl. Default is true
placement	Specify where to place the policy
faultClass	if placement is a FAULT, this specifies which fault the policy would apply to

When using a custom Spring configuration, you need to import META-INF/cxf/cxf-extension-policy.xml

```
@Policies({
    @Policy(uri = "annotationpolicies/TestInterfacePolicy.xml"),
    @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
            placement = Policy.Placement.SERVICE_PORT),
    @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
            placement = Policy.Placement.PORT_TYPE)
}
)
@WebService
public static interface TestInterface {
    @Policies({
        @Policy(uri = "annotationpolicies/TestOperationPolicy.xml"),
        @Policy(uri = "annotationpolicies/TestOperationInputPolicy.xml",
                placement = Policy.Placement.BINDING_OPERATION_INPUT),
        @Policy(uri = "annotationpolicies/TestOperationOutputPolicy.xml",
                placement = Policy.Placement.BINDING_OPERATION_OUTPUT),
        @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION),
        @Policy(uri = "annotationpolicies/TestOperationPTInputPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION_INPUT),
        @Policy(uri = "annotationpolicies/TestOperationPTOutputPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION_OUTPUT)
    }
    )
    int echoInt(int i);
}
```

org.apache.cxf.annotations.UseAsyncMethod (since 2.6.0)

Used on the JAX-WS service implementation object to mark a method as preferring the 'async' version of the method instead of the synchronous version. With JAX-WS, services default to the synchronous methods that require the returning value to be returned from the method. By marking a method with the @UseAsyncMethod annotation, if the transport supports it, CXF will call the async version that takes an AsyncHandler object and the service can call that handler when the response is ready. If the transport does not support the CXF continuations, the synchronous method will be called as normal.

```
    @UseAsyncMethod
    public String greetMeSometime(String me) {
        LOG.info("Executing operation greetMeSometime synchronously");
        System.out.println("Executing operation greetMeSometime synchronously\n");
        return "How are you " + me;
    }

    public Future<?>  greetMeSometimeAsync(final String me,
                                           final AsyncHandler<GreetMeSometimeResponse> asyncHandler) {
        LOG.info("Executing operation greetMeSometimeAsync asynchronously");
        System.out.println("Executing operation greetMeSometimeAsync asynchronously\n");
        final ServerAsyncResponse<GreetMeSometimeResponse> r
            = new ServerAsyncResponse<GreetMeSometimeResponse>();
        new Thread() {
            public void run() {
                GreetMeSometimeResponse resp = new GreetMeSometimeResponse();
                resp.setResponseType("How are you " + me);
                r.set(resp);
                System.out.println("Responding on background thread\n");
                asyncHandler.handleResponse(r);
            }
        } .start();

        return r;
    }
```