Client HTTP Transport (including SSL support)

Default Client Transport

Prior to Apache CXF 3.6.0 / 4.0.1, the default HTTP client transport was **URLConnectionHTTPConduit**, that is built on top of **java.net.HttpURLConnectio** / **javax.net.ssl.HttpsURLConnection** (JDK standard library). Past these releases, the default HTTP client transport has become **HttpClientHTTPConduit** that is using **java.net.http.HttpClient** under the hood (see please JEP 321: HTTP Client).

The **java.net.http.HttpClient** is reported to be considerably slower than **java.net.HttpURLConnection** (see please https://bugs.openjdk.org/browse/JDK-8277519), so there is an option to revert the default to **URLConnectionHTTPConduit** by setting **"force.urlconnection.http.conduit"** contextual property to `true` .

The **java.net.http.HttpClient** is very heavy and hungry for resources (heap and threads primarily). Apache CXF tries to countermeasure that by sharing a single **java.net.http.HttpClient** instance across many HTTP client conduits if possible (subject to HTTP client policy and SSL/TLS settings). This behaviour could be changed by setting `share.httpclient.http.conduit` contextual property to `false` (the default value is `true`), in this case a new **java.net.http.HttpClient** instance is going to be created per HTTP client conduit.

Authentication

Basic Authentication

```
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns="http://cxf.apache.org/transports/http/configuration">
  <authorization>
     <sec:UserName>myuser</sec:UserName>
     <sec:Password>mypasswd</sec:Password>
     <sec:AuthorizationType>Basic</sec:AuthorizationType>
  </authorization>
</conduit>
```

Note: The AuthorizationType element can be omitted if you're using Basic authentication, as above.

Digest Authentication

Same as above but use AuthorizationType "Digest".

Supplying dynamic authorization

You can implement the org.apache.cxf.transport.http.auth.HttpAuthSupplier interface or one of its implementations.

The main method this interface provides is:
public String getAuthorization(AuthorizationPolicy authPolicy, URL currentURL, Message message, String fullHeader);

So you get the HttpAuthPolicy, the service URL, the CXF message and the full Authorization header. The fullHeader is the Authorization Header the server sent after the last try. This way you can implement multi phase authentications. You are expected to return the authorization Header to send to the server. For a simple implementation you can look at org.apache.cxf.transport.http.auth.DefaultBasicAuthSupplier.

If you set your implementation class as AuthSupplier on the conduit CXF will use it.

Spnego Authentication (Kerberos)

Starting with CXF 2.4.0 CXF supports Spnego authentication using the standard AuthPolicy mechanism. Spnego is activated by setting the AuthPolicy.authorizationType to 'Negotiate'. If userName is left blank then single sign on is used with the TGT from e.g. Windows Login. If userName is set then a new LoginContext is established and the ticket is created out of this.

By default the SpnegoAuthSupplier uses the OID for Spnego. Some servers require the OID for Kerberos. This can be activated by setting the contextual property auth.spnego.useKerberosOid to 'true'.

Kerberos Config:

Make sure that krb5.conf/krb5.ini is configured correctly for the Kerberos realm you want to authenticate against
and supply it to your application by setting the java.security.krb5.conf system property

Login Config:

Create a file login.conf and supply it to CXF using the System property java.security.auth.login.config.

The file should contain:

```
CXFClient {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};
```

Sample config:

Make sure the Authorization element contains the same name as the Section in the login.conf (here: CXFClient).

HTTP conduit configuration for spnego with single sign on

```
...
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
  xmlns="http://cxf.apache.org/transports/http/configuration">
  <authorization>
     <AuthorizationType>Negotiate</AuthorizationType>
     <Authorization>CXFClient</Authorization>
  </authorization>
</conduit>
...
```

You can use UserName and Password in the above xml config if you want to log in explicitly. If you want to use the cached Ticket Granting Ticket then do not supply them.

On windows you will also have to make sure you allow the TGT to be used in Java. See: http://www.javaactivedirectory.com/?page_id=93

Switching to Kerberos OID instead of Spnego

```
...
<jaxws:client>
 <jaxws:properties>
  <entry key="auth.spnego.useKerberosOid" value="true"/>
 </jaxws:properties>
</jaxws:client>
...
```

Credential Delegation

Please set an "auth.spnego.requireCredDelegation" property to "true" if you need to enable the credential delegation. Note that setting this property will let the receiving service implement the credential delegation.

If the Kerberos credential is already available in the service request context then one can make this credential available to Spnego/Kerberos authentication handler by setting it on the current CXF message, using an 'org.ietf.jgss.GSSCredential' key.

This can be done before a client invocation is made, by setting a client request context property, or by extending 'org.apache.cxf.transport.http.auth.AbstractSpnegoAuthSupplier'. Please see this thread for more information on the latter option.

Note in the case of reusing the existing credential, the policy configuration does not need to reference a login module name:

HTTP conduit configuration for spnego with single sign on

```
...
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
  xmlns="http://cxf.apache.org/transports/http/configuration">
  <authorization>
     <AuthorizationType>Negotiate</AuthorizationType>
  </authorization>
</conduit>
...
```

NTLM Authentication

CXF doesn't support NTLM authentication "out of the box" on Java 5, but with some additional libraries and configuration, the standard HttpURLConnection objects that we use can do the NTLM authentication. On Java 6, NTLM authentication is built into the Java runtime and you don't need to do anything special.

On Java 5, you need a library that will augment the HttpURLConnection to do it. See: http://jcifs.samba.org/src/docs/httpclient.html Note: jcifs is LGPL licensed, not Apache licensed.

Next, you need to configure jcifs to use the correct domains, wins servers, etc... Notice that the bit which sets the username/password to use for NTLM is commented out. If credentials are missing jcifs will use the underlying NT credentials.

```
//Set the jcifs properties
jcifs.Config.setProperty("jcifs.smb.client.domain", "ben.com");
jcifs.Config.setProperty("jcifs.netbios.wins", "xxx.xxx.xxx.xxx");
jcifs.Config.setProperty("jcifs.smb.client.soTimeout", "300000"); // 5 minutes
jcifs.Config.setProperty("jcifs.netbios.cachePolicy", "1200"); // 20 minutes
//jcifs.Config.setProperty("jcifs.smb.client.username", "myNTLogin");
//jcifs.Config.setProperty("jcifs.smb.client.password", "secret");

//Register the jcifs URL handler to enable NTLM
jcifs.Config.registerSmbURLHandler();
```

Finally, you need to setup the CXF client to turn off chunking. The reason is that the NTLM authentication requires a 3 part handshake which breaks the streaming.

```
//Turn off chunking so that NTLM can occur
Client client = ClientProxy.getClient(port);
HTTPConduit http = (HTTPConduit) client.getConduit();
HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();
httpClientPolicy.setConnectionTimeout(36000);
httpClientPolicy.setAllowChunking(false);
http.setClient(httpClientPolicy);
```

Please also see Asynchronous HTTP Conduit for more information on NTLM.

Proxy Authentication

Proxy authentication can be configured as follows.

```
<conduit name="{http://example.com/}HelloWorldServicePort.http-conduit"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns="http://cxf.apache.org/transports/http/configuration">
  <proxyAuthorization>
     <sec:UserName>myuser</sec:UserName>
     <sec:Password>mypasswd</sec:Password>
  </proxyAuthorization>
  <client AllowChunking="false" ProxyServer="localhost" ProxyServerPort="8080" />
</conduit>
```

This works over HTTPS and HTTPS, but note for the latter it is necessary to set the following system property (see here for more information "Disable Basic authentication for HTTPS tunneling"):

```
-Djdk.http.auth.tunneling.disabledSchemes=
```

Configuring SSL Support

When using an "https" URL, CXF will, by default, use the certs and keystores that are part of the JDK. For many HTTPs applications, that is enough and no configuration is necessary. However, when using custom client certificates or self signed server certificates or similar, you may need to specifically configure in the keystores and trust managers and such to establish the SSL connection.

To configure your client to use SSL, you'll need to add an <http:conduit> definition to your XML configuration file. See the Configuration guide to learn how to supply your own XML configuration file to CXF. If you are already using Spring, this can be added to your existing beans definitions.

A wsdl_first_https sample can be found in the CXF distribution with more detail.

Here is a sample of what your conduit definition might look like:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="
      http://cxf.apache.org/configuration/security
      http://cxf.apache.org/schemas/configuration/security.xsd
      http://cxf.apache.org/transports/http/configuration
      http://cxf.apache.org/schemas/configuration/http-conf.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <http:conduit name="{http://apache.org/hello_world}HelloWorld.http-conduit">

    <http:tlsClientParameters>
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="my/file/dir/Morpit.jks"/>
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore type="JKS" password="password"
                      file="my/file/dir/Truststore.jks"/>
      </sec:trustManagers>
      <sec:cipherSuitesFilter>
        <!-- these filters ensure that a ciphersuite with
             export-suitable or null encryption is used,
             but exclude anonymous Diffie-Hellman key change as
             this is vulnerable to man-in-the-middle attacks -->
        <sec:include>.*_EXPORT_.*</sec:include>
        <sec:include>.*_EXPORT1024_.*</sec:include>
        <sec:include>.*_WITH_DES_.*</sec:include>
        <sec:include>.*_WITH_AES_.*</sec:include>
        <sec:include>.*_WITH_NULL_.*</sec:include>
        <sec:exclude>.*_DH_anon_.*</sec:exclude>
      </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
    <http:authorization>
      <sec:UserName>Betty</sec:UserName>
      <sec:Password>password</sec:Password>
    </http:authorization>
    <http:client AutoRedirect="true" Connection="Keep-Alive"/>

  </http:conduit>

</beans>
```

The first thing to notice is the "name" attribute on <http:conduit>. This allows CXF to associate this HTTP Conduit configuration with a particular WSDL Port. The name includes the service's namespace, the WSDL port name (as found in the wsdl:service section of the WSDL), and ".http-conduit". It follows this template: "{WSDL Namespace}portName.http-conduit". Note: it's the PORT name, not the service name. Thus, it's likely something like "MyServicePort", not "MyService". If you are having trouble getting the template to work, another (temporary) option for the name value is simply "*.http-conduit".

Another option for the name attribute is a reg-ex expression (e.g., "http://localhost:*") for the ORIGINAL URL of the endpoint. The configuration is matched at conduit creation so the address used in the WSDL or used for the JAX-WS Service.create(...) call can be used for the name. For example, you can do:

```
   <http:conduit name="http://localhost:8080/.*">
       ......
   </http:conduit>
```

to configure a conduit for all interactions on localhost:8080. If you have multiple clients interacting with different services on the same server, this is probably the easiest way to configure it.

If your service endpoint uses an SSL WSDL location (i.e., "https://xxx?wsdl"), you can configure the http conduit to pick up the SSL configuration by using a hardcoded http conduit name of "{http://cxf.apache.org}TransportURIResolver.http-conduit". The specific HTTP conduit name or a reg-ex expression can still be used.

Keystores (as identified by the sec:keyStore element above) can be identified via any one of three ways: via a file, resource, or url attribute. File locations are either an absolute path or relative to the working directory, the resource attribute is relative to the classpath, and URLs must be a valid URL such as "http://..." "file:///...", etc. Only one attribute of "url", "file", or "resource" is allowed.

Advanced Configuration

HTTP client endpoints can specify a number of HTTP connection attributes including whether the endpoint automatically accepts redirect responses, whether the endpoint can use chunking, whether the endpoint will request a keep-alive, and how the endpoint interacts with proxies.

A client endpoint can be configured using three mechanisms:

- Configuration
- WSDL
- Java code

Using Configuration

Namespace

The elements used to configure an HTTP client are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix `http-conf`. In order to use the HTTP configuration elements you will need to add the lines shown below to the beans element of your endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

HTTP Consumer Configuration Namespace

```
<beans ...
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration
       ...
       xsi:schemaLocation="...
           http://cxf.apache.org/transports/http/configuration
           http://cxf.apache.org/schemas/configuration/http-conf.xsd
       ...">
```

The `conduit` element

You configure an HTTP client using the `http-conf:conduit` element and its children. The `http-conf:conduit` element takes a single attribute, `name`, that specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form *portQName*`.http-conduit`. For example, the code below shows the `http-conf:conduit` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort>` if the endpoint's target namespace was `http://widgets.widgetvendor.net`. Alternatively, the `name` attribute can be a regular expression to match a URL. This allows configuration of conduits that are not used for purposes of WSDL based endpoints such as JAX-RS and for WSDL retrieval.

http-conf:conduit Element

```
...
  <http-conf:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
    ...
  </http-conf:conduit>

  <http-conf:conduit name="*.http-conduit">
  <!-- you can also using the wild card to specify
       the http-conduit that you want to configure -->
    ...
  </http-conf:conduit>

  <http-conf:conduit name="http://localhost:8080/.*">
  <!-- you can also using the reg-ex URL matching for
       the http-conduit that you want to configure -->
    ...
  </http-conf:conduit>
...
```

The `http-conf:conduit` element has a number of child elements that specify configuration information. They are described below. See also Sun's JSSE Guide for more information on configuring SSL.

Element	Description
`http-conf:client`	Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc.
`http-conf:authorization`	Specifies the the parameters for configuring the basic authentication method that the endpoint uses preemptively.
`http-conf:proxyAuthorization`	Specifies the parameters for configuring basic authentication against outgoing HTTP proxy servers.
`http-conf:tlsClientParameters`	Specifies the parameters used to configure SSL/TLS.
`http-conf:authSupplier`	Specifies the bean reference or class name of the object that supplies the authentication information used by the endpoint both preemptively or in response to a 401 HTTP challenge.
`http-conf:trustDecider`	Specifies the bean reference or class name of the object that checks the HTTP(S) URLConnection object in order to establish trust for a connection with an HTTPS service provider before any information is transmitted.

The `client` element

The `http-conf:client` element is used to configure the non-security properties of a client's HTTP connection. Its attributes, described below, specify the connection's properties.

Attribute	Description
`ConnectionTimeout`	Specifies the amount of time, in milliseconds, that the client will attempt to establish a connection before it times out. The default is 30000 (30 seconds). 0 specifies that the client will continue to attempt to open a connection indefinitely.
`ReceiveTimeout`	Specifies the amount of time, in milliseconds, that the client will wait for a response before it times out. The default is 60000. 0 specifies that the client will wait indefinitely.
`AutoRedirect`	Specifies if the client will automatically follow a server issued redirection. The default is false.
`MaxRetransmits`	Specifies the maximum number of times a client will retransmit a request to satisfy a redirect. The default is -1 which specifies that unlimited retransmissions are allowed.
`AllowChunking`	Specifies whether the client will send requests using chunking. The default is true which specifies that the client will use chunking when sending requests. Chunking cannot be used if either of the following are true: • `http-conf:basicAuthSupplier` is configured to provide credentials preemptively. • `AutoRedirect` is set to true. In both cases the value of `AllowChunking` is ignored and chunking is disallowed. See note about chunking below.
`ChunkingThreshold`	Specifies the threshold at which CXF will switch from non-chunking to chunking. By default, messages less than 4K are buffered and sent non-chunked. Once this threshold is reached, the message is chunked.
`Accept`	Specifies what media types the client is prepared to handle. The value is used as the value of the HTTP `Accept` property. The value of the attribute is specified using as multipurpose internet mail extensions (MIME) types. See note about chunking below.
`AcceptLanguage`	Specifies what language (for example, American English) the client prefers for the purposes of receiving a response. The value is used as the value of the HTTP AcceptLanguage property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
`AcceptEncoding`	Specifies what content encodings the client is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP `AcceptEncoding` property.
`ContentType`	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP `ContentType` property. The default is `text/xml`. **Tip:** For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to application/x-www-form-urlencoded. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to application/octet-stream.
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP `Host` property. **Tip:** This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).

`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • `Keep-Alive`(default) specifies that the client wants to keep its connection open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • `close` specifies that the connection to the server is closed after each request/response sequence.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server.
`Cookie`	Specifies a static cookie to be sent with all requests.
`BrowserType`	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based upon the client that is sending the request.
`Referer`	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. **Note:** This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. **Important:** If the AutoRedirect attribute is set to true and the request is redirected, any value specified in the Refererattribute is overridden. The value of the HTTP Referer property will be set to the URL of the service who redirected the consumer's original request.
`DecoupledEndpoint`	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate server->client connection. **Warning:** You must configure both the client and server to use WS-Addressing for the decoupled endpoint to work.
`ProxyServer`	Specifies the URL of the proxy server through which requests are routed.
`ProxyServerPort`	Specifies the port number of the proxy server through which requests are routed.
NonProxyHosts	Specifies a list of hosts that should be directly routed. This value is a list of patterns separated by '|', where each pattern may start or end with a '*' for wildcard matching.
`ProxyServerType`	Specifies the type of proxy server used to route requests. Valid values are: • HTTP(default) • SOCKS

Example using the `Client` Element

The example below shows a the configuration for an HTTP client that wants to keep its connection to the server open between requests, will only retransmit requests once per invocation, and cannot use chunking streams.

HTTP Consumer Endpoint Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
           http://cxf.apache.org/schemas/configuration/http-conf.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http-conf:client Connection="Keep-Alive"
                      MaxRetransmits="1"
                      AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

Again, see the Configuration page for information on how to get CXF to detect your configuration file.

The `tlsClientParameters` element

Please see TLS Configuration page for more information.

Using WSDL

Namespace

The WSDL extension elements used to configure an HTTP client are defined in the namespace `http://cxf.apache.org/transports/http/configuration`. It is commonly referred to using the prefix `http-conf`. In order to use the HTTP configuration elements you will need to add the line shown below to the `definitions` element of your endpoint's WSDL document.

HTTP Consumer WSDL Element's Namespace

```
<definitions ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration
```

The `client` element

The `http-conf:client` element is used to specify the connection properties of an HTTP client in a WSDL document. The `http-conf:client` element is a child of the WSDL `port` element. It has the same attributes as the `client` element used in the configuration file.

Example

The example below shows a WSDL fragment that configures an HTTP client to specify that it will not interact with caches.

WSDL to Configure an HTTP Consumer Endpoint

```
<service ...>
  <port ...>
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

Using java code

How to configure the HTTPConduit for the SOAP Client?

First you need get the HTTPConduit from the Proxy object or Client, then you can set the HTTPClientPolicy, AuthorizationPolicy, ProxyAuthorizationPolicy, TLSClientParameters.

```
  import org.apache.cxf.endpoint.Client;
  import org.apache.cxf.frontend.ClientProxy;
  import org.apache.cxf.transport.http.HTTPConduit;
  import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;
  ...

  URL wsdl = getClass().getResource("wsdl/greeting.wsdl");
  SOAPService service = new SOAPService(wsdl, serviceName);
  Greeter greeter = service.getPort(portName, Greeter.class);

  // Okay, are you sick of configuration files ?
  // This will show you how to configure the http conduit dynamically
  Client client = ClientProxy.getClient(greeter);
  HTTPConduit http = (HTTPConduit) client.getConduit();

  HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();

  httpClientPolicy.setConnectionTimeout(36000);
  httpClientPolicy.setAllowChunking(false);
  httpClientPolicy.setReceiveTimeout(32000);

  http.setClient(httpClientPolicy);

  ...
  greeter.sayHi("Hello");
```

How to use HTTPConduitConfigurer?

In certain cases, the HTTPConduit could be recreated (for example when using the FailoverFeature) and therefore losing the preconfigured policies. To overcome that, the HTTPConduitConfigurer has been introduced. Here is an example of how it could be used.

```
 HTTPConduitConfigurer httpConduitConfigurer = new HTTPConduitConfigurer() {
    public void configure(String name, String address, HTTPConduit c) {
        HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();

        httpClientPolicy.setConnectionTimeout(36000);
        httpClientPolicy.setAllowChunking(false);
        httpClientPolicy.setReceiveTimeout(32000);

        c.setClient(httpClientPolicy);
    }
}

bus.setExtension(httpConduitConfigurer, HTTPConduitConfigurer.class);
```

How to override the service address ?

If you are using JAXWS API to create the proxy obejct, here is an example which is complete JAX-WS compliant code

```
   URL wsdlURL = MyService.class.getClassLoader
            .getResource ("myService.wsdl");
   QName serviceName = new QName("urn:myService", "MyService");
   MyService service = new MyService(wsdlURL, serviceName);
   ServicePort client = service.getServicePort();
   BindingProvider provider = (BindingProvider)client;
   // You can set the address per request here
   provider.getRequestContext().put(
        BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://my/new/url/to/the/service");
```

If you are using CXF ProxyFactoryBean to create the proxy object , you can do like this

```
   JaxWsProxyFactoryBean proxyFactory = new JaxWsProxyFactoryBean();
   poxyFactory.setServiceClass(ServicePort.class);
   // you could set the service address with this method
   proxyFactory.setAddress("theUrlyouwant");
   ServicePort client = (ServicePort) proxyFactory.create();
```

Here is another way which takes advantage of JAXWS's Service.addPort() API

```
   URL wsdlURL = MyService.class.getClassLoader.getResource("service2.wsdl");
   QName serviceName = new QName("urn:service2", "MyService");
   QName portName = new QName("urn:service2", "ServicePort");
   MyService service = new MyService(wsdlURL, serviceName);
   // You can add whatever address as you want
   service.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://the/new/url/myService");
   // Passing the SEI class that is generated by wsdl2java
   ServicePort proxy = service.getPort(portName, SEI.class);
```

Client Cache Control Directives

The following table lists the cache control directives supported by an HTTP client.

Directive	Behavior

no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, it means the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that will be still be fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive.

A Note About Chunking

There are two ways of putting a body into an HTTP stream:

- The "standard" way used by most browsers is to specify a Content-Length header in the HTTP headers. This allows the receiver to know how much data is coming and when to stop reading. The problem with this approach is that the length needs to be pre-determined. The data cannot be streamed as generated as the length needs to be calculated upfront. Thus, if chunking is turned off, we need to buffer the data in a byte buffer (or temp file if too large) so that the Content-Length can be calculated.
- Chunked - with this mode, the data is sent to the receiver in chunks. Each chunk is preceded by a hexidecimal chunk size. When a chunk size is 0, the receiver knows all the data has been received. This mode allows better streaming as we just need to buffer a small amount, up to 8K by default, and when the buffer fills, write out the chunk.

In general, Chunked will perform better as the streaming can take place directly. HOWEVER, there are some problems with chunking:

- Many proxy servers don't understand it, especially older proxy servers. Many proxy servers want the Content-Length up front so they can allocate a buffer to store the request before passing it onto the real server.
- Some of the older WebServices stacks also have problems with Chunking. Specifically, older versions of .NET.

If you are getting strange errors (generally not soap faults, but other HTTP type errors) when trying to interact with a service, try turning off chunking to see if that helps.

When to set custom headers

If you use a custom CXF interceptor to set one or more outbound HTTP headers then it is recommended to get this interceptor running at a stage preceding the WRITE stage, before the outbound body is written out.

Otherwise the custom headers may get lost. The headers may get retained in some cases even if they are added after the body is written out, example, when a chunking threshold value (4K by default) has not been reached,

but relying on it for the headers not to be lost is brittle and should be avoided.

Asynchronous HTTP Conduit

Please see Asynchronous HTTP Conduit page for more information.