Standalone HTTP Transport

Configuring SSL

To configure the standalone HTTP transport to use SSL, you'll need to add an <http:destination> 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. For more information about configuring TLS, see the Configuring TLS page.

Destinations in CXF are responsible for listening for server side requests.

```
<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:httpj="http://cxf.apache.org/transports/http-jetty/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://cxf.apache.org/transports/http-jetty/configuration
             http://cxf.apache.org/schemas/configuration/http-jetty.xsd
          http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <http:destination name="{http://apache.org/hello_world}GreeterImplPort.http-destination">
  </http:destination>

  <httpj:engine-factory bus="cxf">
  <!-- you just need to specify the TLS Server configuration for the certain port -->
    <httpj:engine port="9003">
      <httpj:tlsServerParameters>
        <sec:keyManagers keyPassword="password">
          <sec:keyStore type="JKS" password="password"
                        file="src/test/java/org/apache/cxf/systest/http/resources/Tarpin.jks"/>
        </sec:keyManagers>
        <sec:trustManagers>
          <sec:keyStore type="JKS" password="password"
                        file="src/test/java/org/apache/cxf/systest/http/resources/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>
        <sec:clientAuthentication want="true" required="true"/>
      </httpj:tlsServerParameters>
    </httpj:engine>
  </httpj:engine-factory>
</bean>
```

Add the static content pages into the jetty server

The CXF standalone http transport is based on the jetty server. The code below shows how to get the jetty server from the destination and how to add the static content path to the jetty server.

```
    // get the jetty server form the destination
    EndpointInfo ei = new EndpointInfo();
    ei.setAddress(serviceFactory.getAddress());
    Destination destination = df.getDestination(ei);
    JettyHTTPDestination jettyDestination = (JettyHTTPDestination) destination;
    ServerEngine engine = jettyDestination.getEngine();
    Handler handler = engine.getServant(new URL(serviceFactory.getAddress()));
    org.mortbay.jetty.Server server = handler.getServer(); // The Server

    // We have to create a HandlerList structure that includes both a ResourceHandler for the static
    // content as well as the ContextHandlerCollection created by CXF (which we retrieve as serverHandler).
    Handler serverHandler = server.getHandler();
    HandlerList handlerList = new HandlerList();
    ResourceHandler resourceHandler = new ResourceHandler();
    handlerList.addHandler(resourceHandler);
    handlerList.addHandler(serverHandler);

    // replace the CXF servlet connect collection with the list.
    server.setHandler(handlerList);
    // and tell the handler list that it is alive.
    handlerList.start();

    // setup the resource handler
    File staticContentFile = new File(staticContentPath); // ordinary pathname.
    URL targetURL = new URL("file://" + staticContentFile.getCanonicalPath());
    FileResource fileResource = new FileResource(targetURL);
    resourceHandler.setBaseResource(fileResource);
```

Configuring Standalone Jetty in OSGi

An OSGi application can use the standalone transport by incorporating cxf-rt-transports-http-jetty, which is an OSGi bundle, and creating services with endpoint URLs like `http://0.0.0.0:8181/rest/v1/entities`.

If your application needs to configure Jetty, is has to communicate with CXF's mechanism for this purpose, which lives in `org.apache.cxf.transport.http_jetty.osgi.HTTPJettyTransportActivator`.

CXF allows an application to use different configuration parameters for different 'engines'. An engine is specified by three properties:

- HTTP vs. HTTPS (protocol)
- host
- port

For each endpoint, CXF allows you to specify:

- sessionSupport
- continuationsEnabled
- reuseAddress
- maxIdleTime
- a giant, complex, raft of TLS parameters for HTTPS.

Each of these engines is a shared resource, in that many endpoints can coexist on an engine. Thus, these shared parameters can't be configured as part of launching an endpoint.

Instead, CXF (in HTTPJettyTransportActivator), creates an OSGi `ManagedServiceFactory`. Usually, a managed service factory is a creature that creates multiple OSGi services on demand, each with a particular configuration.
CXF doesn't really do that. Instead, each time that an application publishes a configuration, CXF sets the parameters for the engine described by the configuration properties.

Concretely, if an application calls the `createFactoryConfiguration` method of the `ConfigurationAdmin` service for PID `org.apache.cxf.http.jetty`, CXF looks in the properties. It expects to find `host` and `port`. If it finds any TLS parameters, it configures for HTTPS. If not, it configures for HTTP. It then applies all the other parameters to the engine for that [protocol,host,port] triple.

If your application needs to create such a configuration and then modify if (all before CXF actually starts up the engine), you must use code like:

```
configurationAdmin.listConfigurations(String.format("(&(service.factoryPid=%s)(%s=%s))",
 parsed[0], ConfigurationFileReaderImpl.INSTANCE_ID, parsed[1]));
```

to locate the configuration you created before, where "ConfigurationFileReaderImpl.INSTANCE_ID" is just an example of how you would use a property of your own choice to uniquely specify the configuration.