Distributed OSGi Reference

Distributed OSGi Reference Guide

- Distributed OSGi Reference Guide
  - Configuration Properties
    - Service Provider properties For Configuring SOAP-based services and consumers
    - Service Provider properties For Configuring RESTful JAXRS-based endpoints and consumers
    - Service Consumer properties
  - Custom intents
  - remote-services.xml files
  - Contributing Distribution properties to Existing Services (without changing them)

CONFIGURATION PROPERTIES

New in DOSGI 1.2: Servlet Filters (javax.servlet.Filter) can be registered as OSGi services with the "org.apache.cxf.httpservice.filter" boolean
property set to true and used to secure DOSGi server endpoints.Endpoints can enforce the registration of the filters by setting an "org.apache.cxf.httpservice.requirefilter" boolean property to true.

These properties are set on the Service Registration in the OSGi Service Registry.

Service Provider properties For Configuring SOAP-based services and consumers

Note: for backwards compatibility old values marked below are still supported.

Property Name	Data Type	Example	Description

service.exported.interfaces (previously:*osgi.remote.interfaces*)	String	`org.example.BarService, org.example.FooService *`	Denotes the interfaces to be exposed remotely. This is a comma-separated list of fully qualified Java interfaces that should be made available remotely. A special value of `*` can be provided meaning that *all* of the interfaces passed to the `BundleContext.registerService()` call are suitable for remoting.
service.exported.configs (previously:*osgi.remote.configuration.type*)	String	`org.apache.cxf.ws`	Specifies the mechanism for configuring the service exposure. Possible values: • `org.apache.cxf.ws` (previously: `pojo`) the OSGi Service is exposed as a Web Service. • `wsdl` configuration driven from WSDL

`org.apache.cxf.ws` configuration type

When the `service.exported.configs=org.apache.cxf.ws` (or `osgi.remote.configuration.type=pojo`) property is specified, the following properties may also be specified.

Property Name	Data Type	Example	Description
org.apache.cxf.ws.address (previously:*osgi.remote.configuration.pojo.address*)	String	http://localhost:9090/greeter	The address at which the service with be made available remotely. If this property is not specified, this defaults to http://localhost:9000/fully/qualified/ClassName.

org.apache.cxf.ws.httpservice.context (previously:*osgi.remote.configuration.pojo.httpservice.context*)	String	`/auction`	When this property is specified, the OSGi HTTP Service is used to expose the service, rather than a dedicated Jetty HTTP Server. This property doesn't allow the specification of a port number, as this is provided by the HTTP Service. The Distributed OSGi distributions come with Pax-Web, for which configuration information can be found at http://wiki.ops4j.org/display/paxweb/Configuration, however other OSGi HTTP Service implementations are potentially configured differently.
org.apache.cxf.ws.frontend	String	`jaxws`	The CXF frontend which will be used to create endpoints. Defaults to 'simple' which is an Aegis-based simple frontend. Note that for JAXWS to work a javax.jws.* has to be imported into the interface and/or implementation and client bundles for annotations like @WebService and @WebMethod be recognized
org.apache.cxf.ws.databinding	String	`jaxb`	Supported values are 'aegis and 'jaxb', defaults to 'aegis'. Note that for JAXB to work JAXB packages like javax.xml.bind.annotation.* have to be imported
org.apache.cxf.ws.databinding.bean	DataBinding		An actual DataBinding instance to use. If not specified, a default one is created according to the type specified in the org.apache.cxf.ws.databinding property.

org.apache.cxf.ws.wsdl.location	String	`/wsdl/service.wsdl`	WSDL location
org.apache.cxf.ws.service.ns	String	http://services.org	WSDL service namespace
org.apache.cxf.ws.service.name	String	`SoapService`	WSDL service name
org.apache.cxf.ws.port.name	String	`SoapServicePort`	WSDL port name
org.apache.cxf.ws.in.interceptors	String, String[], List		List of CXF in interceptors
org.apache.cxf.ws.out.interceptors	String, String[], List		List of CXF out interceptors

org.apache.cxf.ws.in.fault.interceptors	String, String[], List		List of CXF in fault interceptors
org.apache.cxf.ws.out.fault.interceptors	String, String[], List		List of CXF out fault interceptors
org.apache.cxf.ws.features	String, String[], List, Object		List of CXF out features

Service Provider properties For Configuring RESTful JAXRS-based endpoints and consumers

`org.apache.cxf.rs` configuration type

When the `service.exported.configs=org.apache.cxf.rs` property is specified, the following properties may also be specified.

Property Name	Data Type	Example	Description
org.apache.cxf.rs.address	String	http://localhost:9090/greeter	The address at which the service with be made available remotely. If this property is not specified, this defaults to http://localhost:9000/fully/qualified/ClassName.

org.apache.cxf.rs.httpservice.context	String	`/auction`	When this property is specified, the OSGi HTTP Service which is used to expose the service, rather than a dedicated Jetty HTTP Server. By default, absolute address may look like 'http://localhost:8080/auction'
org.apache.cxf.rs.provider	Boolean	`true/false`	Can be used to identify a global JAXRS provider as CXF-compatible
org.apache.cxf.rs.provider.expected	Boolean	`true/false`	Can be used to require global providers to set an 'org.apache.cxf.rs.provider' property with a value 'true'.
org.apache.cxf.rs.provider.globalquery	Boolean	`true/false`	Can be used to disable queries for global providers, defaults to 'true'.
org.apache.cxf.rs.databinding	String	`aegis`	This property has a limited value for JAXRS services as JAXB is supported by default, the only supported value is 'aegis' and it is a shortcut for registering an Aegis provider, see below for more information on how to register custom providers for JAXRS services
org.apache.cxf.rs.wadl.location	String	`/wadl/service.wadl`	WADL location

org.apache.cxf.rs.provider	String, String[], List		List of JAX-RS providers
org.apache.cxf.rs.in.interceptors	String, String[], List		List of CXF in interceptors
org.apache.cxf.rs.out.interceptors	String, String[], List		List of CXF out interceptors
org.apache.cxf.rs.in.fault.interceptors	String, String[], List		List of CXF in fault interceptors
org.apache.cxf.rs.out.fault.interceptors	String, String[], List		List of CXF out fault interceptors
org.apache.cxf.rs.features	String, String[], List		List of CXF out features

Note that by default for JAXRS to work javax.ws.rs.* packages have to be imported into the interface and/or implementation and client bundles for annotations like @Path and @Context be recognized. You can avoid importing JAXRS annotations if you provide an out-of-band model. The way it is done in a greeter_rest demo is described here. The model files can be located in a OSGI-INF/cxf/jaxrs resource folder and can be named as model.xml or ServiceName-model.xml (ex : GreeterService-model.xml).
If you use JAXB and you would like to avoid importing JAXB packages into your application bundles then you can try registering a custom JAXB provider which is configured as described here.

Registering custom JAXRS providers

Custom JAXRS providers including CXF-specific providers can be registered like regular OSGI services, for example :

```
Object provider = new CustomMessageBodyReaderWriter();
bundleContext.registerService(
  new String[]{"javax.ws.rs.ext.MessageBodyReader", "javax.ws.rs.ext.
MessageBodyReader"}, provider);
```

Note that when registering a global provider, one may set an 'org.apache.cxf.rs.provider.expected' on a given service description thus requiring providers to confirm that they will reliably work with CXF JAX-RS by setting a 'org.apache.cxf.rs.provider' true property during the registration - this may be needed when multiple JAX-RS implementations are available and some custom providers depending on JAXRS implementation specific code.

Alternatively, one can register per-service specific providers during the application service registration :

```
CustomMessageBodyReaderWriter provider1 = new CustomMessageBodyReaderWriter();
provider.setCustomProperty(true);
CustomMessageBodyReaderWriter provider2 = new CustomMessageBodyReaderWriter();
provider2.setCustomProperty(false);

Dictionary properties = new Hashtable();
properties.put("org.apache.cxf.rs.provider", provider);

Dictionary properties2 = new Hashtable();
properties.put("org.apache.cxf.rs.provider", provider2);


bundleContext.registerService(
  new String[]{"org.books.BookService"}, new BookServiceImpl(), properties);
bundleContext.registerService(
  new String[]{"org.books.BookService"}, new AdvancedBookServiceImpl(),
properties2);
```

Finally, one can declare them using "org.apache.cxf.rs.provider" :

```
<property name="org.apache.cxf.rs.provider" value="org.foo.bar.Provider1,org.
foo.bar.Provider2"/>
```

or, when using declarative services :

```
<property name="org.apache.cxf.rs.provider">
  org.foo.bar.Provider1
  org.foo.bar.Provider2
</property>
```

Service Consumer properties

On client side proxies, typically the same properties are set as on set service provider side for both SOAP and RESTful clients. There are some additional properties too. Since the client-side proxy is registered by the DOSGi implementation, all these properties are read-only.

Property Name	Data Type	Example	Description
service.imported	boolean	`true`	This property is always set on a service proxy, indicating that the real service is remote.
org.apache.cxf.remote.dsw.client	String		This property is set to the bundle name of the CXF-DOSGi implementation and can be used to find client side proxies created by the CXF DOSGi implementation.

CUSTOM INTENTS

Intents allow to define custom configurations for DOSGi services. In the service exports the intents are listed by name in the property "service.exported.intents".

In version 1.4.0 and above custom intents are defined as OSGi services. The property name "org.apache.cxf.dosgi.IntentName" is used to mark the service as an intent. The intent name value then can be used to reference the intent in OSGi services. Custom intents can either be CXF Features or a CXF Binding Configuration.

`REMOTE-SERVICES.XML` FILES

The CXF DOSGi implementation provides a DSW (Distribution Software) implementation of Distributed OSGi. It is compatible with any Distributed OSGi Discovery implementation in order to discover remote services dynamically.

However, using a Discovery system is optional, it is also possible to statically configure remote services into the system. This is done by registering one or more bundles containing `remote-services.xml` files. By default the system looks for any files with the `.xml` extension in the `OSGI-INF/remote-service` directory of the bundle.

Here's an example:

```
<service-descriptions xmlns="http://www.osgi.org/xmlns/sd/v1.0.0">
  <service-description>
    <provide interface="org.apache.cxf.dosgi.samples.greeter.GreeterService" />
    <property name="osgi.remote.interfaces">*</property>
    <property name="osgi.remote.configuration.type">pojo</property>
    <property name="osgi.remote.configuration.pojo.address">http://localhost:
9090/greeter</property>
  </service-description>

  <!-- further service-description tags are allowed here -->
</service-descriptions>
```

Alternative locations

By default all `*.xml` files in the OSGI-INF/remote-service location are considered, this location can be changed by setting the `Remote-Service` header in the bundle manifest, e.g.

```
Remote-Service: META-INF/osgi
```

CONTRIBUTING DISTRIBUTION PROPERTIES TO EXISTING SERVICES (WITHOUT CHANGING THEM)

@@@ TODO check that this still works with the 1.2 release.

CXF/DOSGi allows you to add the distribution properties to existing OSGi services. You can do this by installing a bundle that contains an XML file with the extra properties in the `OSGI-INF/remote-service` directory:

A sample `OSGI-INF/remote-service/sd.xml` file looks like this:

```
<service-decorations xmlns="http://cxf.apache.org/xmlns/service-decoration/1.
0.0">
 <service-decoration>
   <match interface="org.apache.F(.*)">
     <match-property name="test.prop" value="xyz"/>
     <add-property name="service.exported.interfaces" value="*"/>
   </match>
 </service-decoration>
</service-decorations>
```

A service decorations file can have any number of `service-decoration` tags, each tag describing a **match** rule for services that are to be decorated.
The match rules are defined as follows:

- `match interface="org.apache.Foo"` matches any service that is registered under the *org.apache.Foo* class or interface. The `interface` attribute takes regular expressions, so specifying `org.apache(.)*` will match any service registered with an interface in a subpackage of *org.apache*.

- The optional `match-property` tags allows you to declare extra conditions to be applied to services of which the interface matches. In the above example the rule will only match services that have the `test.prop` property set to the value `xyz`. Other services don't match. Any number of `match-property` tags can be specified.
- The `add-property` specifies the extra property to be added to the remote service. The above example adds `service.exported.interfaces="*"` which will cause any matching service to be exposed remotely. The `add-property` has an optional `type` attribute which defaults to `java.lang.String`. You can specify other Java basic types such as `java.lang.Long` if needed. You can have any number of `add-property` tags.

Note the bundle with the extra metadata will need to be started before the bundle with the service that is to be remoted is started (need to fix this).