AXIS2 ESSENTIAL ADMINISTRATOR GUIDE

Introduzione

Il documento descrive le nozioni essenziali, o richieste più frequentemente, necessarie per configurare Axis2 su Tomcat all'interno di un ambiente di produzione distribuito.

Sommario

1. Configurazione

Logging

Il sistema di log di Axis2 è configurato nel file `log4j.properties`.
Se Axis2 è installato come server stand-alone questo file potrebbe trovarsi in:

`/axis2-1.4.1` oppure `/axis2-1.4.1/webapp/WEB-INF/classes`

Se Axis2 è installato come web app all'interno di Tomcat il file dovrebbe trovarsi qui:

`/apache-tomcat/webapps/axis2/WEB-INF/classes/` :

Non è ben chiaro quale di questi percorsi abbia la precedenza sull'altro (la questione sembra dipendere sia dall'installazione di Axis2, sia dal sistema operativo).

Front End

Per cambiare l'indirizzo del front end associato ad Axis2 vedere il parametro **httpFrontendHostUrl** nel file `axis2.xml`.

2. Soap address e WSDL location

L'impiego del campo `soap:address` è specificato dal consorzio W3C, che recita:

> *The SOAP address binding is used to give a port an address (a URI). A port using the SOAP binding MUST specify exactly one address. The URI scheme specified for the address must correspond to the transport specified by the soap:binding*

(tratto da `http://www.w3.org/TR/wsdl#_soap:address`)

Perciò l'unica constrains mandatory è quella di indicare un singolo URI (i.e. non più di uno) il quale deve corrispondere al binding definito dalla sezione `soap:binding`. Ad esempio

```
<wsdl:binding name="my_project_SOAP" type="tns:main_port">
```

quindi il WSDL è corretto, almeno dal punto di vista sintattico, anche se riporta un campo *location* inesistente, poiché soddisfa comunque entrambi i requisiti specificati dal W3C. Dal punto di vista semantico, invece, indicare un indirizzo fittizio (e.g. "http://www.example.org/") può apparire strano o fuorviante.
Questa scelta è però spesso conveniente per i seguenti motivi:

1) Solitamente uno stesso web service (WS) viene deployato in più ambienti (test e produzione), che hanno sempre EPR diversi. Se il campo *location* dovesse rispettare l'URI effettivo del WS (ovvero l'EPR), si avrebbe un WSDL diverso per ogni ambiente, cosa che in alcuni casi può risultare scomoda o creare confusione.

2) E' possibile configurare l'Application Server affinché il campo *location* venga valorizzato dinamicamente, a runtime, con il valore dell'EPR. Questo consentirebbe di avere il campo *location* sempre allineato con l'EPR relativo all'ambiente di deploy effettivo.
Tale funzionalità è però utile solamente quando l'Application Server viene raggiunto direttamente, ad esempio perché esposto sulla rete pubblica o perché raggiungibile via VPN. Quando invece l'EPR si trova dietro un Front End, si possono avere dei protocolli di comunicazione eterogenei tra gli attori coinvolti.

CLIENT <--- https ---> FRONT END <--- http ---> APPLICATION SERVER

In questo caso si dovrebbe inserire un valore del tipo *location="https ://some_uri"* nel WSDL da deployare sull'AS, in modo che il WSDL ritornato al client dal Front End contenga il valore corretto, ovvero l'indirizzo del web service come se fosse sul Front End. Così facendo si avrebbe però che all'avvio dell'AS egli penserebbe che il WS debba comunicare via HTTPS, e se non trova l'abilitazione necessaria (il modulo SSL è spesso disabilitato sull'AS, visto che NON deve essere usato) lancia un'eccezione del tipo "modulo https non abilitato". Conseguentemente il WS non viene neppure deployato, perché inconsistente con la configurazione dell'AS che lo ospita.

Una soluzione potrebbe essere quella di inserire nel WSDL un campo *location* simile a quello effettivo (ovvero "`http://some_uri`" invece di "`https ://some_uri`"), ma ciò genera ancora più confusione, perché in questo caso il client potrebbe invocare il WS usando il protocollo sbagliato (http invece di https), evento che provocherebbe la mancata autenticazione sul FE.

3) Anche risolvendo il problema dei diversi protocolli (ad esempio abilitando la comunicazione https tra FE e AS e trovando poi il modo di non usarla), l'AS fornirebbe dinamicamente nel campo *location* l'URI effettivo del WS, che in questo caso sarebbe un URI locale del tipo:

`location="http://120.12.12.42:8080/axis2/services/nome/"` (IP locale dell'AS)

e non quella effettiva del FE, che sarebbe invece qualcosa del tipo

`location="https://frontend.adriani.net/webservices/nome/"` (URL pubblico del FE)

e quindi risulterebbe fuorviante, errato e addirittura pericoloso per la sicurezza!

Conclusioni

Il problema potrebbe essere risolto studiando una configurazione ad hoc tra AS e FE, oppure applicando opportune mappature degli indirizzi, ma solitamente si preferisce evitare tale lavoro poiché non strettamente necessario, dato che è molto più semplice comunicare un unico WSDL per tutti gi ambienti di deploy e poi, a parte, comunicare la lista degli EPR pubblici per raggiungere le varie versioni del WS (test e produzione).

3. Date e Time Zone

Se un web service deve utilizzare un elemento XML contenente una data, è necessario capire come gestire la definizione dei fusi orari e dell'eventuale ora legale, anche se il client e il server si trovano entrambi in Italia.
Per questo motivo illustriamo brevemente come Java gestisce gli oggetti SOAP *Date* e *Time Zone*.
Quando la macchina utilizza l'ora legale le date sono espresso per mezzo di etichette del tipo:

CET o CEST	*Central European Time* o *Central European Summer Time*
IST o IDT	*Indian Standard Time* o *Israel Daylight Time*
etc...	

questo a seconda che la data in esame sia relativa al periodo dell'ora legale (*daylight saving*) o no.
Se la data viene convertita in un oggetto Java `Calendar`, essa verrà memorizzata senza alcuna conversione.
Al momento di estrarre la data mediante il metodo `calendar.getTime()` si otterrà invece un valore che dipende da tre variabili:

- I **valori** base indicati nell'oggetto `calendar` originale.
  Esempio: per la stringa "2010-04-23T11:30:00+02:00" i valori base sono le 11:30 del 23/04/2010.

- L'**offset** indicato nell'oggetto `calendar` originale.
  Esempio: per la stringa "2010-04-23T11:30:00+02:00" l'offset è 7.200.000 (2 ore).

- Il **time zone** configurato sulla macchina locale.
  Esempio: nel time zone italiano la stringa "2010-04-23T11:30:00+02:00" indica un periodo di vigore dell'ora legale, per cui tale data verrà considerata in *daylight saving mode*.

La data prodotta dal metodo `calendar.getTime()` viene determinato come segue:

1. Si sottrae l'**offset** al **valore base**, recuperando questi dati dal `calendar` stesso, in modo da trovare la data in formato UTC.

2. Si aggiusta il risultato in base al **raw offset** del time zone associato al `calendar`. Nel caso dell'Italia tale valore è fissato a 3.600.000 (1 ora), quindi le ore aumentano di un'unità.

3. Si esprime il risultato secondo il **time zone** associato al `calendar`. In particolare, se l'ora cade nel periodo dell'ora legale di quel time zone, si somma un'ora e si specifica l'etichetta CEST, altrimenti l'ora rimane invariata ed espressa come CET.

Vediamo alcuni esempi riferiti al time zone italiano:

1) La data "2010-02-26T15:40:00+00:00", verrà tradotta come "Fri Feb 26 16:40:00 CET 2010".
   (sommare il raw offset)

2) La data "2010-02-26T15:40:00+01:00" verrà tradotta come "Fri Feb 26 15:40:00 CET 2010".
   (sottrarre l'offset e sommare il raw offset)

3) La data "2010-04-26T15:40:00+01:00" verrà tradotta come "Mon Apr 26 16:40:00 CEST 2010".
   come prima, con l'aggiunta dell'ora legale)

Conclusioni

Anche se si rimane in Italia, **è necessario specificare** nel messaggio SOAP le date "invernali" usando il suffisso `+01:00` e le date "estive" con il suffisso `+02:00`, per evitare errori al momento dell'estrazione della data dall'oggetto `calendar` fornito dalle classi `ADBBean`.

Macchina configurata per ignorare l'ora legale

In questo caso le date saranno espresse col formato `GMT+hh:00`, ad esempio "Fri Feb 26 10:30 GMT+01:00 2010", il cui valore dipenderà solamente dalla data in esame e non dal time zone settato localmente sulla macchina. Con una configurazione come questa le date vanno convertite così:

- Per ottenere l'orario standard UTC: sottrarre l'offset indicato nella data. In questo esempio ("Fri Feb 26 10:30 GMT+01:00 2010") abbiamo +01, per cui secondo l'orario UTC saranno le ore 09:30 del 26 febbraio 2010.

- Per ottenere l'orario locale: aumentare di un'unità le ore se durante il periodo l'ora legale, altrimenti lasciare l'orario immutato. Nel caso "Tue Giu 15 10:30 GMT+01:00 2010" sarebbero le ore 11:30 locali, che infatti corrispondono al tempo GMT+02:00 rispetto all'orario standard UTC.
  Nel caso invece dell'orario visto sopra ("Fri Feb 26 10:30 GMT+01:00 2010"), essendo fuori dal periodo dell'ora legale, l'ora locale coincide con quella indicata, quindi saranno le ore 10:30 locali.

4. Sicurezza

L'installazione dei certificati nell'ambiente Java è governata dallo standard JSSE e viene gestita per mezzo dell'utility a linea di comando **keytool**, che è un tool standard della JDK.

Per importare un **certificato Self Signed** si utilizza il comando

```
keytool -import -keystore my_keystore.jks -storepass 12345678 -file my_cert.cer
```

che importa il certificato `my_cert.cer` all'interno del *keystore* associato al file `my_keystore.jks`.
Se il file non esiste esso viene creato nella directory corrente. Il *keystore* sarà protetto dalla password, che in questo esempio è `12345678`.
Se non viene specificato alcun alias, il *keytool* assegnerà l'alias `mykey` (di default) al certificato. Ciò risulta sconveniente perché al momento di importare un secondo certificato nel *keystore* il *keytool* proverà nuovamente ad assegnare lo stesso alias di default (i.e. `mykey`), e andrà in errore perché gli alias devono essere unici. Per questo motivo è preferibile importare il certificato usando un comando del tipo

```
keytool -import -alias mine.cer -keystore my_keystore.jks -storepass 12345678 -
file my_cert.cer
```

che importa il certificato `my_cert.cer` associandolo all'alias `mine.cer` (all'interno del *keystore*).
Il comando andrebbe eseguito solamente quando occorre importare un nuovo certificato, ad esempio perché il certificato precedente è scaduto. Spesso risulta comodo eseguire il comando direttamente nella directory dov'è installato il programma client, ad esempio `/home/myuser/mysw`.
In tal caso il file creato sarà

```
/home/myuser/mysw/my_keystore.jks
```

Ovviamente è necessario lanciare il comando usando una password forte (e non `12345678`).

Per visualizzare i certificati presenti nel keystore si utilizza il comando

```
keytool -list -keystore my_keystore.jks
```
(verrà richiesta la password)

per maggiori informazioni eseguire il comando aggiungendo l'opzione `-v` (verbose).

E' buona norma eliminare i certificati vecchi o scaduti dal keystore, ad esempio

```
keytool -delete -alias mykey -keystore my_keystore.jks
```

rimuove il certificato associato all'alias `mykey` dal keystore associato al file `my_keystore.jks`.

Dopo aver installato il certificato è sufficiente indicare all'ambiente JSSE il file associato al *keystore* che l'applicazione client dovrà utilizzare a runtime, ad esempio

```
String keystore = "/home/myuser/mysw/my_keystore.jks" ;
…
System.setProperty("javax.net.ssl.trustStore"              , keystore) ;
System.setProperty("javax.net.ssl.trustStorePassword"      , "1234568") ;
System.setProperty("javax.net.ssl.trustStoreType"          , "JKS") ;
```

Nel caso di applicazioni *client* di web services conviene specificare queste informazioni all'interno del file di configurazione dell'applicazione client (ad esempio `httpsConfig.ini`), collocato all'interno della directory di installazione della client stessa.

Nota: se il certificato è garantito da un authority (CA), l'ambiente JSSE dovrebbe riconoscerlo e connettersi automaticamente al server. Per verificare la lista delle authority configurate nel *keystore* lanciare il comando

```
keytool -list -keystore cacerts -storepass changeit
```

dalla directory di lavoro del *keytool* (ad esempio: `jdk/jre/lib/security`).

