AXIS2 QUICK START

1. Introduzione

Apache-Axis2 implementa le API di JAX-WS (`javax.jws.*`) per semplificare lo sviluppo e la distribuzione di Java Web Services (JWS) basati sul protocollo SOAP. La versione precedente di Axis utilizzava le librerie JAX-RPC, sviluppate sul modello *stub-skeleton* ispirato dal paradigma RMI.
Axis2 può lavorare come server stand-alone, ma risulta conveniente installarlo come webapp su Application Server per delegare all'AS la gestione delle request HTTP, in modo da sfruttarne robustezza e scalabilità.
Nel caso di Apache-Tomcat per installare Axis2 come webapp occorre copiare il file `axis2.war` all'interno della cartella `webapps` e riavviare Tomcat.

2. Deployment

Un Web Service per Axis2 va organizzato secondo la seguente struttura (l'etichetta `root` è solo d'esempio):

- `root/`: copiare qui la struttura delle cartelle contenenti classi e risorse.
- `root/lib`: eventuali librerie o drivers (JAR files).
- `root/META-INF`: deve contenere il file `services.xml` e il documento WSDL.

Invocando il comando `jar -f` (eseguito dalla VM) all'interno della cartella `root`, specificando come target `<nome servizio>.aar`, si produce l'archivio AAR da copiare nel percorso:

```
<TOMCAT_HOME>\webapps\axis2\WEB-INF\services
```

Per esporre il servizio non è necessario riavviare Tomcat (Axis2 supporta l'*hot deployment* per default).
In alternativa è possibile collegarsi all'indirizzo `http://localhost:8080/axis2/` e autenticarsi nella sezione **administration** (usando login e password indicati nel file `\axis2\WEB-INF\conf\axis2.xml`).
Ciò permette di usare il tool *Upload Service* (che non fa altro che copiare l'AAR nel percorso già menzionato).
E' possibiledeployare il servizio anche in formato "esploso" (i.e. senza creare l' AAR) semplicemente copiando il contenuto della cartella `root` all'interno del percorso `axis2\WEB-INF\services\<`*nome servizio*`>`. In ogni caso l'endpoint del servizio sarà qualcosa del tipo:

```
http://localhost:8080/axis2/services/<nome del servizio>
```

3. Configurazione

La **configurazione** del servizio è specificata dal file `META-INF/services.xml` :

```
<serviceGroup>
  <service name="myService">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-out"
            class="adriani.webapp.soap.MyServiceMessageReceiverInOut"/>
    </messageReceivers>
    <parameter name="ServiceClass">adriani.webapp.soap.MyServiceSkeleton</parameter>
  </service>
</serviceGroup>
```

Il nome del servizio (*myService*) va indicato come attributo della tag `service`. A seconda del MEP (Message Exchange Pattern) utilizzato occorre specificare la classe che si occupa di ricevere/inoltrare i messaggi. Infine si indica il nome della classe che implementa il servizio vero e proprio. Notare che il trasporto HTTP offerto da Axis2, per default, gestisce indifferentemente le richieste SOAP e HTTP, per cui è sempre possibile invocare il servizio attraverso una semplice HTTP GET, qualsiasi sia il WSDL associato.

4. Aggiornamento

Per aggiornare un servizio già deployato su Axis2 è buona norma chiudere Tomcat, cancellare il vecchio AAR e poi procedere con l'installazione della nuova versione. In alternativa è possibile impostare il parametro `hotdeployment` a `true` nel file di configurazione:

```
\apache-tomcat\webapps\axis2\WEB-INF\conf\axis2.xml
```

In tal modo caricando un servizio per mezzo del tool *Upload Service* si sostituisce la versione precedente.
Tale scelta è però sconsigliata, in quanto potrebbe lasciare il sistema in uno stato non prevedibile.
Esempio: se il servizio utilizza degli oggetti statici o lancia dei thread di supporto, tali entità potrebbero rimanere "attive" anche dopo l'aggiornamento del servizio.

5. Contesto di esecuzione

Axis2 utilizza cinque diversi tipi di contesto per gestire la persistenza dello stato, ciascuno dei quali ha lo stesso *lifetime* dell'entità a cui è riferito (ad esempio il `MessageContext` esiste solo per il tempo necessario a processare il messaggio a cui fa riferimento):

- `MessageContext`: stato del singolo messaggio.
- `OperationContext`: stato della singola operazione (metodo); eventualmente può contenere diversi *MessageContext* contemporaneamente (ingresso ed uscita).
- `ServiceContext`: lo stato runtime del servizio, la cui durata dipende perciò dalla `scope` selezionato per il servizio (vedi sotto).
- `ServiceGroupContext`: utile nel caso si decida di raggruppare assieme servizi diversi.
- `ConfigurationContext`: la configurazione permanente di Axis2.

6. Scope e multi-threading

Axis2 assegna sempre un nuovo thread (preso da un *pool*) per ogni invocazione del servizio. Le richieste del servizio vengono processate nello stesso ordine in cui arrivano sul *server*. Notare che se il client lavora in modo asincrono, oppure non garantisce che chiamate consecutive avvengano in un ordine ben preciso, potrebbe risultare un ordine d'esecuzione "apparentemente diverso" tra le chiamate lato client e le risposte lato server.
La persistenza del servizio invece dipende dal parametro `scope` del file `services.xml`:

- `request`: viene creato un nuovo oggetto per ogni invocazione del web service (**default**).

- `application`: tutte le invocazioni utilizzano lo stesso oggetto (ognuna col suo thread).

- `soapsession`: il server risponde alla prima chiamata inserendo un elemento *ServiceGroupId* nella header della risposta SOAP. Se la client specifica tale ID nella header di una richiesta successiva, Axis2 permette di gestire una sorta di sessione, memorizzando i dati necessari in un oggetto `ServiceContext` oppure `ServiceGroupContext` (a seconda che la transazione coinvolga un solo servizio o un gruppo di servizi). Tale "sessione" ha spesso un timeout piuttosto breve (il timeout di default dovrebbe essere pari a 30 secondi).

- `transportsession`: Axis2 memorizza gli oggetti `ServiceContext` e `ServiceGroupContext` nella sessione associata al trasporto utilizzato (ad esempio una sessione HTTP). Ciò consente di condividere i dati della sessione anche tra gruppi diversi, ma la durata della sessione dipende dal gestore del trasporto e non da Axis2. Per attivare la funzione occorre modificare il parametro `manageTransportSession` nel file `axis2.xml`.

7. Sviluppo su Eclipse

Come ambiente di sviluppo faremo riferimento ad Eclipse for Java EE Developers, i.e. Eclipse con il plugin Web Tools Platform. Occorre inanzitutto verificare la configurazione di Eclipse:

- *Preferences – Server – Runtime Enviroments*: Apache Tomcat.
- *Preferences – Web Services – Axis2 Preferences*: indicare l'ubicazione della root di Axis2.

A questo punto è possibile:

- Creare un progetto del tipo *Dynamic Web Project*.
- Aggiungere il progetto appena creato usando *Add and remove projects* sul server Tomcat.

Sviluppo Top-Down

Lo sviluppo Top-Down consiste nel produrre il codice di un web service partendo dal documento WSDL. Esistono molti tool che realizzano questo compito. Su Eclipse si può procedere così:

1. Creare un nuovo file WSDL in una qualsiasi cartella del progetto: *New - Web Services – WSDL*.

2. Editare il file WSDL (vedasi il documento "03_WSDL_quick_start.pdf" per dettagli).

3. Lanciare il wizard di creazione del web service: *New - Web Services – Web Services*.
   Scegliere la voce *Top down …*, indicare l'ubicazione del file WSDL e posizionare la slide in alto (*Test service*). In caso di errore del wizard chiudere Eclipse e controllare che tra i processi in background non sia rimasto attiva una precedente sessione di Eclipse.

4. Implementare lo skeleton del servizio affinché venga ritornato l'oggetto previsto dal WSDL (la SOAP response).

8. Struttura ADB

Eclipse genera tutte le classi previste dal modello ADB (Axis Data Binding), ovvero:

- `<nome servizio>Skeleton.java`: la classe "main" del servizio.
- `<nome servizio>MessaggeReceiverXY.java`: da 1 a 4 classi che si occupano di gestire il trasporto nei 4 casi di MEP possibili (InOut, OutIn, In e Out).
- `ExtensionMapper.java`: mappa le estensioni (i.e i sotto elementi) dei messaggi XML nei relativi oggetti Java.

Per risalire al nome del servizio dal nome di queste classi occorre fare attenzione alla capitalizzazione, ad esempio. *MyServiceSkeleton* è la classe principale del servizio avente nome *myService*.

Oltre a queste classi, Eclipse genera una classe per ogni *complex type* definito nel WSDL.

Nota: per default il build di Eclipse compila le classi ed eventualmente genera il corrretto file AAR, ma **non copia** i file di configurazione (XML e WSDL) all'interno del percorso `services/<myService>/META-INF`. Perciò, se questi file vengono modificati, può essere necessario copiarli a mano per poter testare il servizio.

9. Web Services Explorer

E' una client generica di Eclipe che permette di testare un web service usando i vari tipi di binding supportati. Per aprire la client cliccare sull'icona *Launch the Web Services Explorer* (solitamente vicina ad *Open Web Browser*) e procedere come riportato qui sotto:

1. Cliccare sull'icona `WSDL page` (in alto a destra, vicino all'icona dei "favoriti"), selezionare la voce `WSDL Main` nella finestra *Navigator* e inserire nella finestra *Actions* l'endpoint del WSDL descrittore del servizio, che dovrebbe essere qualcosa del genere:

   `http://localhost:8080/<`**nome webapp**`>/services/<`*nome del servizio*`>?wsdl`

   (è meglio non usare URL con indirizzo IP esplicito, per evitare problemi con l'eventuale proxy)

2. Aprire il menù *Endpoints* (cliccando su ▼), cliccare su *Add* e inserire un URL del tipo:

   `http://localhost:8080/<`**nome webapp**`>/services/<`*nome del servizio*`>`

   cliccare su *Go* per salvare il nuovo endpoint.

3. Cliccare sul nome dell'operazione (metodo) desiderato, scegliere il corretto endpoint, inserire il valore dei parametri d'ingresso ed infine cliccare su *Go*.

Nota 1: per default Eclipse utilizza Axis2 come modulo di una webapp (i.e. inserendo la cartella `axis2-web` allo stesso livello di WEB-INF), per cui il *nome webapp* è quello assegnato dallo sviluppatore alla webapp. Se invece il web service è deployato direttamente in Axis2, allora il *nome webapp* è `axis2`.

Nota 2: se il messaggio d'ingresso è un *simple type*, come ad esempio:

```
<param0 type="xsc:string">valore</param0>
```

Il Web Services Explorer non permette di inserire il valore all'interno di un *textfield*: in tal caso occorre cliccare su **Add** affianco al nome del parametro per far comparire il campo (sprovvisto di etichetta). Questo non è necessario se il messaggio d'ingresso è invece un *complex type*, come ad esempio:

```
<myElement><param0 type="xsc:string">valore</param0></myElement>
```


10. Axis2 Trouble Shooting

In questa sezione elenchiamo alcuni dei problemi più ricorrenti che possono sorgere quando ci si avvicina ad Axis2 per la prima volta. La lista qui sotto è riferita alla distribuzione Axis2 1.4.x.

- `The service cannot be found for the endpoint reference (EPR)`: il servizio richiesto è inesistente, oppure l'endpoint utilizzato per invocare il servizio è sbagliato.

- `The endpoint reference (EPR) for the Operation not found`: il servizio richiesto è corretto, ma il nome dell'operazione invocata è sbagliato.

- `namespace mismatch require`: il valore (non il prefisso) del namespace usato per gli elementi della SOAP request è diverso da quello dichiarato come *targetNameSpace* nel WSDL.

- `ADBException: Unexpected subelement`: la request SOAP contiene elementi in meno (o in più) rispetto a quelli previsti. Oppure il namespace usato per gli elementi della SOAP request è diverso da quello usato dal server per costruire la SOAP response.

- `AxisFault ... at MessaggeReceiver`: la request SOAP è errata o incompleta (mancano degli elementi attesi come parametri d'ingresso). Questo errore può verificarsi anche nel caso che i parametri d'ingresso non soddisfino le costraints definite nel WSDL.

  Nota: Axis2 non gestisce correttamente gli attributi XSD del tipo `nillable` e `minoccurs` nella Request SOAP (vedi sotto), perciò il problema potrebbe essere dovuto ad una tag dichiarata opzionale nel WSDL, mentre Axis2 si aspetta tale elemento come se fosse obbligatorio.

- *La request SOAP non gestisce l'opzione minoccors*: se un WSDL prevede un elemento con `minOccurs="0"`, le classi prodotte nell'approccio top-down potrebbero non gestire tale elemento, che risulterà perciò obbligatorio e non opzionale. *Soluzione*: specificare il valore `maxOccurs="1"` esplicitamente nel WSDL del servizio.

- *Risposta SOAP non coerente con il WSDL*: Axis2 prevede una validazione solo parziale[1] della risposta SOAP rispetto allo schema XSD definito nel WSDL. Ciò significa che è possibile implementare un web service che fornisce una risposta diversa da quella descritta dal WSDL. Spesso questo l'errore è dovuto al processo di sviluppo del codice, che dovrebbe sempre coerente con il WSDL a priori, non a posteriori.

- *Elemento XML del tipo <element_name xsi:nil="1">*: il modello ADB restituisce tale elemento quando il codice Java invoca un metodo `setXXX(null)` sull'oggetto `ADBBean` in questione, che deve essere definito come *nillable*. Se invece il codice non valorizza affatto tale oggetto, allora si ottiene in risposta un elemento "vuoto" del tipo `<element_name/>` (questo se le specifiche del WSDL consentono la presenza della stringa vuota nell'elemento, altrimenti si avrà una violazione delle constraints).

[1] Ad esempio, Axis2 non da errore se in uscita vi sono degli elementi XML in più rispetto a quelli dichiarati nel WSDL. Axis2 controlla però che quelli coerenti siano non nulli (se non dichiarati come `nillable`).

11. soapUI Trouble Shooting

soapUI è un applicativo che può risultare comodo al momento di verificare il funzionamento di un web service. La comodità di soapUI consiste nel fatto che il programma genera il client del web service automaticamente, in modo visuale, partendo semplicemente dall'endpoint del servizio. Inoltre, sin dalla versione free soapUI consente di preparare delle unità di test robuste e flessibili, configurabili mediante *Groovy Script*.
Il programma è reperibile all'indirizzo:

http://www.soapui.org/

Di seguito sono elencate alcune delle problematiche più frequenti quando si muovono i primi passi con il programma:

- *Il servizio non è raggiungibile*: SoapUI (come molte altri client) deduce l'indirizzo dell'endpoint dal WSDL del servizio. Tale metodo in genere non funziona se la connessione al web service avviene attraverso un proxy o mediante una connessione SSL. In tal caso l'indirizzo del WSDL è raggiunbile, ma per invocare il servizio è necessario precisare manualmente l'endpoint.
  Esempio: supponiamo di specificare in SoapUI un indirizzo del tipo:

  ```
  https://myhost.net/webservices/myService?wsdl
  ```

  supponiamo che il WSDL venga letto ma il servizio non risponda: probabilmente occorre sostituire tale indirizzo con l'endpoint vero e proprio del metodo da invocare, ad esempio:

  ```
  https://myhost.net/webservices/myService/myOperation
  ```

- *La risposta soap viene trasmessa correttamente, ma non arriva al client*: se una o più definizioni dei *data types* XSD contenute nel WSDL sono state cambiate, ma l'interfaccia è rimasta la stessa, SoapUI continua ad invocare il web service usando i *data types* del vecchio WSDL. In tal caso è necessario aggiornare l'interfaccia lato client ricaricando il WSDL (cliccare sull'interfaccia del servizio e scegliere *update definition*).

