1-feb-10
rev. 0.1
SA

DESIGN WEB SERVICE TEMPLATE

Sommario

1. Introduzione

Il presente documento descrive l'architettura del software *template*, realizzato utilizzando la tecnologia Java sulla piattaforma Tomcat con il modulo Axis2. Il software costituisce un web service completo e funzionante, ma sprovvisto di una qualsiasi *business logic* applicativa. Il software *template* si presta quindi come "scheletro di partenza" per implementare un qualsiasi web service.

2. Packages

Le classi che costituiscono il servizio sono organizzate nei seguenti packages:

- `adriani.jws.template`: contiene solamente la classe `Settings`, dove viene memorizzata la configurazione generale del web service.
- `adriani.jws.template.client`: contiene alcune classi eseguibili da console per testare il servizio a linea di comando, nonché alcune classi generiche (*esempio*: test di alcune features di Java5).
- `adriani.jws.template.db`: gestisce le connessioni al database e le classi DBA.
- `adriani.jws.template.eclipse`: contiene le classi `ADBBean` generate da Eclipse a partire dal documento WSDL (sviluppo *top down*).
- `adriani.jws. template.messagges`: i *wrappers* usati per avvolgere le classi `ADBBean` allo scopo di garantire il loose coupling tra il documento WSDL e l'implementazione del servizio. Il package definisce anche le eccezioni lanciate in caso di mancata validazione dei messaggi SOAP.
- `adriani.jws.template.service`: implementa la *business logic* vera e propria del servizio, la gestione dei messaggi d'errore, i parametri di funzionamento e l'analisi delle performances.
- `adriani.jws.template.util`: contiene le classi usate per gestire il meccanismo di log e la gestione dei vari *thread*.

3. Caricamento statico

Alla prima invocazione del servizio vengono eseguiti alcuni blocchi di codice statici: ogni blocco si occupa di caricare i parametri ed instanziare gli oggetti relativi solamente a una particolare funzione.

- **Database**: carica dal file `configuration.properties` **5** parametri d’uso generale, più **4** parametri per ogni database richiesto dal web service. Dopodiché instanzia gli oggetti `JDBCConnectionPool` che gestiscono i pool di connessione ai database.

- **GeneralService**: carica **5** parametri dal file `configuration.properties` relativi al funzionamento generale del web service (flags booleani di vario tipo). Qui viene inizializzato l’oggetto `WSLogger` per gestire il meccanismo di log e l’oggetto `ServiceData` per caricare i parametri di funzionamento della *business logic*.

- **WSLogger**: carica **6** parametri dal file `configuration.properties` relativi al meccanismo di log.

- **ServiceData**: carica i parametri di funzionamento della logica del servizio: l’eventuale serie di chiavi e/o valori dal file `keywords.properties` e i parametri (dal file `configuration.properties`) che modificano il comportamento della *business logic*.

- **ErrorMessagge**: carica i messaggi d’errore dal file `messages_list.properties`. La versione attuale forza il caricamento del file `messages_list_en.properties` per garantire il comportamento uniforme del web service in caso di Locale non previsti.

4. Architettura

Database (db)

La classe `Database` si occupa principalmente di eseguire un blocco statico che carica i parametri di **configurazione JDBC** (solamente la prima volta che la classe viene caricata), il tempo di **timeout** utilizzato dal `CleanUpThread` e la **dimensione** massima del pool. All'interno del blocco statico vengono create le istanze dei pool di connessione e memorizzate come attributi statici della classe `Database`. Tutte le DBA utilizzate dal web service dovrebbero richiedere le connessioni JDBC con un codice del tipo:

```
Connection connection = Database.getJDBCPool(key).getConnection() ;
…
// Eseguire qui le operazioni SQL richieste
…
Database.getJDBCPool(key).returnConnection(connection) ;
```

Il parametro `key` permette di selezionare il pool associato al database desiderato.
Esempio: se il web service deve accedere a tre database diversi, il parametro `key` può assumere i valori 1, 2 o 3 (mappati nei valori statici `DB_01`, `DB_02` e `DB_03` della classe `Database`).

La classe `ServiceDBA` utilizza un sistema leggermente diverso, poiché un'invocazione del servizio potrebbe richiedere tipicamente una decina di query SQL da eseguire in rapida successione. Per questo motivo la `ServiceDBA` utilizza il metodo `openConnection()` per richiedere una connessione dal pool (come visto sopra) e memorizza il puntatore a tale connessione come attributo di classe.
In ogni caso è compito del layer superiore (`ServiceMacro` oppure `ServiceLogic`) chiedere e restituire la connessione al pool utilizzando i metodi `openConnection()` e `closeConnection()`.

Gli **indirizzi dei database** possono essere specificati in due modi: indicando l'indirizzo IP del database server nel file `configuration.properties` (ad esempio `10.1.5.42:1233`) oppure indicando l'alias **tns** (ad esempio `my_schema`) sempre nello stesso file. Se presente, l'alias tns ha la precedenza sull'indirizzo IP.
Il path del file `tnsnames.ora` dei nomi tns dei database è indicato dalla chiave `oracle.tnsnames`, e solitamente è del tipo:

```
/u01/app/oracle/product/10.2.0/network/admin
```

Tale percorcorso può essere specificato nel file `configuration.properties`.

La classe astratta `ObjectPool` è la parent class della `JDBCConnectionPool`.
Essa offre alcuni metodi per gestire un generico pool di oggetti, che viene suddiviso tra due `Hashtables`: una per gli oggetti *locked* (utilizzati dal servizio) e una per gli oggetti *unlocked* (disponibili).
La sottoclasse `JDBCConnectionPool` implementa i metodi astratti di `ObjectPool` creando un **pool di connessioni** che vengono richieste per mezzo della chiamata JDBC:

```
DriverManager.getConnection()
```

l'helper class `CleanUpThread`, contenuta in `ObjectPool`, implementa un `Thread` che si occupa di ripulire il pool di oggetti periodicamente, chiamando il metodo `ObjectPool.cleanUp()`. Indicativamente il *thread* dovrebbe attivarsi ogni cinque minuti (il parametro è configurabile nel file `configuration.properties`).

Il package contiene anche la classe `MockConnectionPool`, che essendo un'estensione della `JDBCConnectionPool` può essere utilizzata in alternativa al pool JDBC. La `MockConnectionPool` eredita quasi interamente il comportamento della `JDBCConnectionPool`, ma offre due nuovi metodi non presenti nella parent class:

- `MockConnection getMockConnection() throws SQLException`
- `returnMockConnection(MockConnection mock)`

La differenza principale è che il `JDBCConnectionPool` gestisce oggetti `java.sql.Connection`, mentre il `MockConnectionPool` gestisce oggetti del tipo `MockConnection`, che sono dei wrappers delle `java.sql.Connection`. Se il parametro `pool.mocked` è settato a YES la classe `Database` utilizzerà il "mocked pool" anziché quello JDBC standard, che dovrebbe essere utile nei seguenti casi:

- E' necessario testare e/o debuggare il comportamento degli oggetti associati alle connessioni SQL. Ad esempio, se si vogliono avere informazioni su un `PreparedStament`, questo può essere memorizzato all'interno del wrapper `MockConnection` (che contiene già una `Hashtable` con tale scopo).

- Il driver JDBC non implementa la precompilazione dei prepared statement, che perciò vengono re-instanziati ogni volta. In tal caso potrebbe essere utile memorizzare gli statement all'interno della `MockConnection` per migliorare le perforamances.

A parte questi casi, in linea di massima, è sempre meglio utilizzare la classe `JDBCConnectionPool`.

Per garantire la massima trasparenza tra i due meccanismi alle classi DBA, la classe Database offre i metodi `getMockConnection`() e `returnMockConnection`() che ritornano sempre un oggetto `MockConnection`, anche quando il pool è in realtà un normale pool JDBC.

Service

Contiene le classi che implementano la *business logic* del servizio.

La classe `GeneralService` si occupa di inizializzare l'oggetto `WSLogger` che gestisce il **meccanismo di log** utilizzato da tutte le altre classi. Per questo motivo la classe `GeneralService` non viene mai instanziata, e viene caricata solo la prima volta che viene invocato il servizio vero e proprio (i.e. una sottoclasse).

La classe `ServiceMacro` gestisce le chiamate alle necessarie DBA. Qui andrebbe implementata la *business logic* di basso livello, corrispondente al *middle layer* del web service.

La classe `ServiceData` si occupa di caricare e memorizzare la lista degli eventuali parametri (ad esempio una lista di prefissi telefonici) dal file `keywords.properties`. Essa offre inoltre il metodo `createKeywords()` utile per creare un file del tipo `keywords.properties` usando i valori caricati nella classe a runtime.
Questa classe viene inizializzata dal blocco statico contenuto nella `GeneralService`.

La classe `ServiceLogic` rappresenta la business logic di alto livello del web service (*top layer*).
In linea teorica questa classe NON dovrebbe mai accedere alle DBA, ma dovrebbe delegare alla `ServiceMacro` l'accesso ai dati. In tal modo la richiesta e il rilascio delle connessioni al database dovrebbe restare relegato al *middle layer*.

Il package contiene anche la classe `ServiceAnalyzer` che raccoglie alcuni dati sulle performances del servizio. Questa classe viene instanziata da `GeneralService` solamente se il parametro di configurazione `service.analyze` è pari a YES. I comandi per la raccolta di questi dati sono spiegati in appendice.

Eclipse

Qui vengono inserite le classi prodotte dal *Eclipse Web Platform Tool* (WTP) in base al WSDL del servizio (sviluppo *top down*). La maggior parte di queste classi implementano l'interfaccia `ADBBean` (Axis2 DataBinding Framework). Se fosse necessario modificare il WSDL è possibile generare delle nuove classi, sempre per mezzo del WTP e sostituire completamente le classi di questo package.

Nota: se la modifica del WSDL è di lieve impatto la classe `TemplateSkeleton` potrebbe rimanere invariata, per cui si consiglia di effettuare sempre una copia di backup di questa classe, prima di sovrascriverla (per non perdere l'eventuale documentazione inclusa della classe).

Se le modifiche del WSDL riguardano solamente la struttura "interna" degli elementi XML dei messaggi SOAP (*constrains* o *patterns* dei vari *types*) non è necessaria nessun altra modifica. Se invece il nuovo WSDL modifica il nome degli elementi XML, oppure aggiunge/rimuove elementi XML ai messaggi SOAP, allora è necessario aggiornare i corrispondenti wrappers nel package `messages` (vedi sotto).

Messages

La funzione principale di questo package è quella di offrire dei *wrapper* per le classi `ADBBean` che rappresentano i messaggi SOAP (vedi sopra). Le classi `ADBBean` sono a loro volta dei wrapper, ma si è ritenuto opportuno creare ugualmente un ulteriore livello di astrazione per ridurre l'impatto sul software di un eventuale modifica del WSDL o della base dati (*loose coupling*).
Ad esempio, la classe **User** generata dal WSDL è associata al wrapper `WrapperIn`, e viene utilizzata solamente in tre punti:

1. All'interno del package `it.sirti.jws.eclipse` (per costruzione).
2. All'interno della class wrapper `WrapperIn` (per definizione di wrapper).
3. All'interno di `SimpleClient` (per testare il servizio via riga di comando).

Lo stesso dicasi per tutti gli altri messaggi e i relativi wrapper, che elenchiamo per completezza:

- `it.sirti.jws.eclipse.User` → `WrapperIn`
- `it.sirti.jws.eclipse. SingleRecordType` → `WrapperData`
- `it.sirti.jws.eclipse. Information` → `WrapperDataOut`

Le API principali offerte dai wrappers di questo package sono:

- `WrapperData()`: unico costruttore del wrapper, il quale invoca il metodo `setDefaults()` per caricare i valori di default negli attributi di classe.

- `getElement()`: ritorna l'oggetto `ADBBean` avvolto dal wrapper.

- `ADBBean validate()`: metodo principale del wrapper che si occupa di trasferire i valori settati come attributi di classe negli oggetti `ADBBean` (prodotti da Eclipse). Nel caso di *costrains* fissate dal documento WSDL, le corrispondenti classi `ADBBean` potrebbero lanciare delle eccezioni quando il valore non è conferme alle specifiche. In questo caso il wrapper prima assegna il valore di default (solo per quel campo) poi lancia l'eccezione `WrapperXXXFault` per segnalare il problema alla businness logic.

Per garantire la massima flessibilità del software la risposta SOAP del servizio è associata a due wrapper:

- `WrapperData`: avvolge un singolo record, ovvero una serie di elementi XML posti tutti allo stesso livello. Questo wrapper contiene perciò un attributo di classe per ogni elemento `ADBBean` mappato nel wrapper.

- `WrapperDataOut`: memorizza come attributo di classe un array di oggetti `WrapperData[]`, che vengono poi assemblati insieme dal metodo `validate()` per formare la risposta SOAP vera e propria. Nel caso di un servizio che ritorna sempre una risposta "singola" (senza alcun array di record), la classe `WrapperDataOut` conterrà perciò un array di `WrapperData` costituito da un unico elemento.

In questo modo è possibile trasformare un web service a risposta singola in un web service che risponde con un array di record senza dover modificare l'intera architettura.

Il package contiene anche le classi `WrapperXXXFault` che raccolgono i dati delle eventuali eccezioni generate dai wrappers stessi, ad esempio durante la validazione di un elemento `ADBBean`.

Util

Contiene un gruppo eterogeneo di classi che implementano il meccanismo di log e la gestione degli eventuali threads.

Il **meccanismo di log** viene inizializzato all'interno del blocco statico della classe `GeneralService`, che invoca il metodo

```
WSLogger.initialize()
```

Il web service offre due tipi di log esclusivi tra loro: quello effettuato da una classe Java (`SystemLogger`) e log effettuato dalla libreria Log4j (`ExternalLogger`). Il tipo di log è stabilito dal valore della chiave *logger.type* nel file di configurazione.
Il log offerto dalla classe Java `SystemLogger` è configurabile nel file `configuration.properties`.
I files di log sono creati all'interno della directory definita dal parametro *logger.dir* posta nel percorso definito dal parametro *path.windows* o *path.linux* (a seconda del sistema operativo). Il percorso assoluto così costruito viene memorizzato nell'attributo `logPath` della classe `SystemLogger`.
I log file prodotti dal `SystemLogger` avranno nome del tipo:

```
d11_h9_m56.txt
```

La rotazione dei file di log è stabilita dal valore della chiave *logger.rotate*.
La classe `LoggerManager` offre anche la possibilità di settare la variabile `logPath` a run-time tramite il metodo `setLocationPath()`, ma tale modalità di lavoro è fortemente sconsigliata. Per i dettagli si rimanda alle API Specifications (JavaDoc) della classe `SystemLogger`.

Il sistema di log offerto da Log4j è implementato dalla classe `ExternalLogger`. Questa classe non sarebbe necessaria per gestire Log4j, ma è stata inserita per mantenere la possibilità di scegliere tra i due meccanismi di log. Il funzionamento di Log4j è completamente configurato nel file `log4covaddress.properties`.

I due sistemi di logger implementano entrambi l'interfaccia `LoggerManager`. Inoltre, per garantire la piena interoperabilità tra i due meccanismi di log, la particolare istanza del `LoggerManager` (che può essere di tipo Java oppure Log4j) è memorizzata come attributo statico della classe `WSLogger`. In tal modo, ogni qualvolta che una classe abbisogna di un proprio logger object, è possibile richiedere il logger corretto definendo un attributo del tipo:

```
static WSLogger logger = WSLogger.getLogger(this.class)
```

sul quale invocare poi i metodi necessari: `logger.logInfo(msg)`, `logger.logWarning(msg)`, `logger.logServer(msg)` oppure `logger. isInfoEnabled()`.

Se la funzionalità **benchmark** viene attivata (*service.benchmark = YES*) i tempi d'esecuzione dei metodi della classe `ServiceDBA` vengono misurati e loggati su file. Questo meccanismo utilizza sempre e comunque il sistema di log offerto dalla classe `SystemLogger` (Java internal log).
I tempi di esecuzione delle chiamate alle DBA sono scaricati nella directory individuata dalla concatenazione dei parametri *path.windows* (o *path.linux*) e *logger.dir*, come già discusso sopra. Il nome dei log files della funzionalità benchmark sarà qualcosa del tipo:

```
analysis_27_11_17.txt
```

Infine, la classe `ThreadManager` permette di monitorare i thread relativi al web service, ma dovrebbe essere usata solamente durante lo sviluppo o la fase di test.

5. Gestione errori

CASO	TRACCIA	MESSAGE	ANALISI
(none)	- getEmptyResponse(SILENT, COMMAND)	RECEIVED REMOTE COMMAND	endCall(COMMAND) command++
`list == ?` (Exception)	- handleMacroError() - getEmptyResponse(SILENT, MACRO)	MACRO ERROR	endCall(MACRO) macro_error++
`list == null` (no data)	- getEmptyResponse(INFO, NO_RESULT)	NO RESULT	endCall(NO_RESULT) no_result++
`list != null` (no database)	- handleNoDatabase() - getEmptyResponse(SILENT, NO_DB)	NO DATABASE	endCall(NO_DB) no_db++
`output == null`	- getEmptyResponse(ERROR, LOGIC)	LOGIC ERROR	endCall(LOGIC) logic_error++
`list != null` (data ok) (but not valid)	- handleNoValidation() - forceMessage() - forceStatus()	VALIDATION ERROR	endCall(VALIDATION) validation_error++
`list != null` (data ok) (and valid)	(normal flow)	NO MESSAGE	endCall(OK) no_error++

Dove il methodo getEmptyResponse() esegue sempre e comunque i seguenti steps:

- getWrapperOut()
- setDefaults()
- validate()
- forceStatus()
- forceMessage()

6. Configurazione

All’interno del file AAR sono presenti i seguenti file di configurazione:

- `META-INF/services.xml`: file di configurazione per Axis2.
- `META-INF/template.wsdl`: descrittore del web service esposto al pubblico.
- `configuration.properties`: file di configurazione principale del web service.
- `messages_list_XX.properties`: supporto multilanguage dei messaggi d’errore.
- `keywords.properties`: lista degli eventuali parametri di funzionamento (place holder).
- `log4template.properties`: configurazione del sistema di log.

Il web service richiede le seguenti librerie:

- **Log4j**: `log4j-1.2.15.jar`.
- **Driver JDBC**: `ojdbc14.jar` e `orai18n.jar`.
- **Axis2**: `axiom-api-1.2.7.jar`, `axis2-adb-1.4.1.jar`, `commons-logging-1.1.1.jar`.

7. Remote Manager

Durante la fase di test può essere utile monitorare lo status del web service, ciò è possibile invocando il servizio usando come primo paramentro la chiave `000000n` (sei zeri seguiti da un codice numerico). In base al codice numerico inviato viene attivato uno dei seguenti comandi:

CODICE	COMANDO
1	Stampa nel file di LOG la **configurazione** attuale del web service
2	Stampa nel file di LOG alcuni dati sull’ambiente di esecuzione **run-time**
3	Stampa nel file di LOG la lista dei **threads** relativi al processo
4	Stampa nel file di LOG alcuni dati sulle connessioni ai **database**
5	Stampa nel file di LOG i risultati della **analisi** e statistiche di funzionamento
6	Setta on/off la modalità di **Analisi** (parametro `Settings.ANALYZE`)
7	Setta on/off la modalità **Verbose** (parametro `Settings.DETAILS`)
8	Setta on/off la modalità **Debug** (parametro `Settings.DEBUG`)
9	Setta on/off la modalità **Remote Manager** (parametro `Settings.MANAGER`)
10	Setta on/off la modalità **Benchmark** (parametro `Settings.BENCH`)
11	Stampa nel file di LOG lo status del **pool JDBC** che gestisce le connessioni ai DB

Importante

Inviando il comando **9** la funzione Remote Manager viene disattivata, per cui l’unico modo di riattivarla è ricaricare il web service sull’application server (questo anche per motivi di sicurezza).

8. Guidelines

La versione base del web service *template* costituisce un web service molto semplice, caratterizzato da un'unica operazione

```
Information getInformation(User user)
```

descritta nel documento WSDL dal blocco

```
<wsdl:portType name="main_port">
      <wsdl:operation name="getInformation">
            <wsdl:input message="tn:service_request"/>
            <wsdl:output message="tn:service_response"/>
      </wsdl:operation>
</wsdl:portType>
```

Come già visto, gli elementi SOAP `User` e `Information` sono mappati rispettivamente nelle classi `WrapperIn` e `WrapperDataOut`. Un web service generico potrebbe però offrire più di un'operazione, caratterizzata da altri elementi in ingresso o in uscita. In tal caso occorre aggiungere le opportune classi nel package `messagges`, tenendo conto delle seguenti linee guida:

1. **Risposta semplice**: il set di elementi rappresentanti la risposta dell'operazione sarà mappata in una classe `WrapperXXX`, la quale deve estendere `Wrapper`.

   Esempio: la classe `WrapperMoney` per la risposta ad un'operazione `getMoney()`.

2. **Risposta completa**: comprenderà un campo `status`, un campo `message` e uno o più occorrenze della risposta semplice. La risposta completa sarà mappata in una classe WrapperXXXOut, la quale deve estendere `WrapperOut`.

   Esempio: la classe `WrapperMoneyOut` per la risposta ad un'operazione `getMoney()`.

3. **Wrappers Factory**: modificare il metodo `getWrapperOut(int method)` della classe `ServiceLogic` aggiungendo all'interno dello `switch` la creazione dei nuovi wrappers.

   Esempio: `case MONEY: wrapper = new WrapperMoneyOut().`

4. **Business logic**: aggiungere i metodi necessari alle classi `ServiceLogic` e `ServiceMacro`.

   Esempio:

   ServiceLogic: `public WrapperMoneyOut getMoney(WrapperIn input)`
   ServiceMacro: `public ArrayList getMoney(WrapperIn input)`

5. **WrapperIn**: in generale i wrappers delle request SOAP d'ingresso non hanno una struttura comune, per cui non esiste una parent class da estendere per i wrapper del tipo `WrapperIn`. La gestione di ulteriori wrappers d'ingresso è perciò completamente libera.

Appendice

Command-line test suit

Specificando nella variabile `CLASSPATH` le librerie necessarie (vedi sotto) è possibile eseguire il comando

```
java -cp %CLASSPATH% it.sirti.jws.template.client.SimpleClient -h
```

che visualizza i comandi riconosciuti a linea di comando dalla classe `SimpleClient`.
In particolare è possibile:

- Simulare il web service senza utilizzare il trasporto dei messaggi SOAP, allo scopo di verificare il funzionamento della business logic senza doversi appoggiare né a Tomcat né a SoapUI.
  Opzione: `-a` (*actual*)

- Verificare il funzionamento della business logic senza utilizzare le classi `ADBBean`, invocando direttamente i metodi della classe `ServiceLogic`.
  Opzione: `-e` (*emulate*)

Se non precisato, il web service viene invocato usando il valore di default **a** (*actual*). La risposta ottenuta viene stampata direttamente a video. Se il servizio viene invocato in modalità *emulazione* i dati della risposta vengono scaricati anche nel file di log (questo perché nella modalità *actual* i dati della risposta sono già disponibili via HTTP).

La classe `SimpleClient` offre anche altre funzioni, non correlate allo sviluppo dei web services, che perciò non sono illustrate nel presente documento.

Esempio di configurazione dell'ambiente richiesto dalla SimpleClient

```
SET CLASSPATH =
  C:\Develop\Build

SET LOG4J_HOME =
  C:\Programmi\Java\jdk1.6.0_12\lib\log4j-1.2.15.jar

SET JDBC_HOME =
  C:\Programmi\Java\jdk1.6.0_12\lib\ojdbc14.jar;
  C:\Programmi\Java\jdk1.6.0_12\lib\orai18n.jar

SET AXISCLASSPATH =
  %AXIS2_HOME%\lib\axiom-api-1.2.7.jar;
  %AXIS2_HOME%\lib\axis2-adb-1.4.1.jar;
  %AXIS2_HOME%\lib\commons-logging-1.1.1.jar;
  %AXIS2_HOME%\lib\axis2-kernel-1.4.1.jar

SET CLASSPATH =
  %CLASSPATH%;
  %LOG4J_HOME%;
  %JDBC_HOME%;
  %AXISCLASSPATH
```