Realizzare unità personalizzate in WebRatio[1]

Le unità personalizzate *(custom unit)*

Lavorando con WebRatio ho trovato la documentazione a volte carente ed incompleta, ho deciso quindi di prendere nota di tutte le informazioni che ho raccolto durante lo sviluppo del lavoro, fornite soprattutto dal gruppo di supporto di WebModels, sempre gentili e disponibili ad assecondare le mie richieste.

In questa parte del lavoro cercherò dunque di descrivere il procedimento per creare nuove unità personalizzate che può essere utile a chi dopo di me vorrà cimentarsi nella creazione di nuove unità. Sebbene ispirato alla documentazione disponibile cercherò qui di omettere particolari non essenziali, rimandando alla guida per gli approfondimenti, e di approfondire invece quelli che ritengo gli aspetti più importanti, non dando per scontato la conoscenza di Eclipse e di WebRatio[2] in modo da poter fornire informazioni, spero utili, a chi si cimenta per la prima volta nella creazione di una unità personalizzata. Molte delle informazioni che ho raccolto in queste pagine non sono presenti nella documentazione e negli esempi ufficiali, derivano invece da risposte a domande, dubbi e difficoltà riscontrate dagli utilizzatori di WebRatio e discusse nei forum di supporto, molto vivi ed attivi.

La documentazione esistente è molto completa da un punto di vista formale, ed elenca con precisione tutte le funzioni disponibili. Non è però facile per un principiante capire tutti i concetti presentati e soprattutto comprendere come legarli tra loro per ottenere il risultato voluto. Questo è un peccato perché le unità personalizzate sono un elemento fondamentale per integrare funzionalità particolari in WebRatio e, una volta padroneggiate le basi, non è così complesso costruire unità personalizzate in poco tempo.

L'idea iniziale di questo scritto è nata nel tentativo personale di cercare di fare chiarezza nella documentazione esistente, spesso così dettagliata nel descrivere i singoli componenti al punto di perdere la visione di insieme e l'aspetto pratico. I pochi esempi disponibili rendono a volte difficile adattare la propria idea ad una nuova unità.

In queste pagine cerco di presentare i concetti che ho sperimentato nella maniera più semplice possibile focalizzandomi più che altro sul risultato desiderato e proponendo uno solo

---

[1] Estratto da: Emmanuele De Andreis, **Pattern per la realizzazione di comunità Web 2.0 con WebRatio** - Politecnico di Milano, luglio 2010

[2] Tutte le schermate e le procedure presentate sono state realizzate e testate usando la versione 5.1.1 di WebRatio

Emmanuele De Andreis – manudea@duemetri.net

dei molteplici modi per realizzarlo, quello più semplice e più immediato. Una volta compresi i passaggi base sarà poi molto semplice approfondire tutte le sfumature e le potenzialità del linguaggio consultando la guida e le risorse disponibili.

Introduzione

WebML, il linguaggio di modellazione di WebRatio, è costruito su pochi elementi altamente componibili, che possono essere utilizzati per assemblare complesse applicazioni Web.

L'aspetto chiave di WebML è la capacità di definire un modello costituito da pagine Web, unità di contenuto e le unità di gestione, tutti legati tra loro. Tuttavia, l'unità di base fornite in WebML potrebbero non essere sufficienti per coprire l'intero spettro di requisiti applicativi, soprattutto nel caso in cui sia necessario usare componenti software esterni nello sviluppo delle proprie applicazioni web. A questo scopo WebRatio comprende il concetto di unità personalizzate (chiamate anche unità plug-in). Un'unità personalizzata (*custom unit*) è un'unità di tipo contenuto (*content unit*) o di tipo operazionale (*operation unit*) definita interamente dallo sviluppatore. E' possibile inserire in un diagramma ipertestuale tutte le unità personalizzate, collegarle ad altre unità e definirne le proprietà; se l'unità è un'unità di contenuto, è anche possibile disporla sulla griglia, e scegliere un modello (*template*) per visualizzare il contenuto. È possibile implementare le unità per fare praticamente qualsiasi cosa: inviare email, pubblicare documenti XML, per interagire con i servizi web.

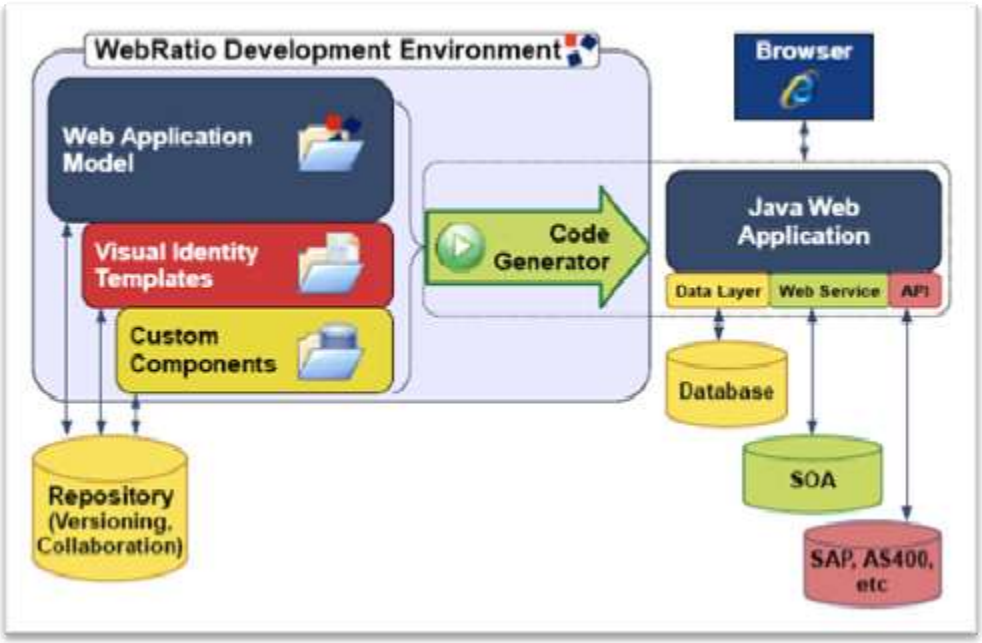

Figura 0-1 – L'architettura interna di un progetto realizzato con WebRatio

Emmanuele De Andreis – manudea@duemetri.net

Per cominciare

L'architettura generale di WebRatio distingue due componenti principali:

- L'ambiente di sviluppo WebRatio, che supporta la fase di progettazione, sviluppo e generazione dell'applicazione
- WebRatio Runtime, che offre tutte le funzionalità necessarie all'esecuzione di applicazioni Java Web generate da WebRatio.

Costruire una unità personalizzata non è una operazione semplicissima perché richiede dunque di padroneggiare diversi concetti che riguardano non soltanto la programmazione java, e soprattutto la programmazione java per il web (javabeans e jsp), ma anche soprattutto una profonda conoscenza di WebRatio e del suo funzionamento interno.

L'unità personalizzata che creeremo è un po' più complessa del tutorial "Hello World" presente all'interno della guida in linea dell'applicativo alla quale rimando per la descrizione più dettagliata delle operazioni elementari.

Tutte le unità personalizzate che l'utilizzatore di WR può creare devono essere incluse in un progetto di tipo speciale ("Units Project").

Si parte aprendo WebRatio e creando un nuovo progetto. Il tipo di progetto è di tipo "Units Project". All'interno del progetto va creata una nuova Unità (New → Unit) all'interno della cartella Units.

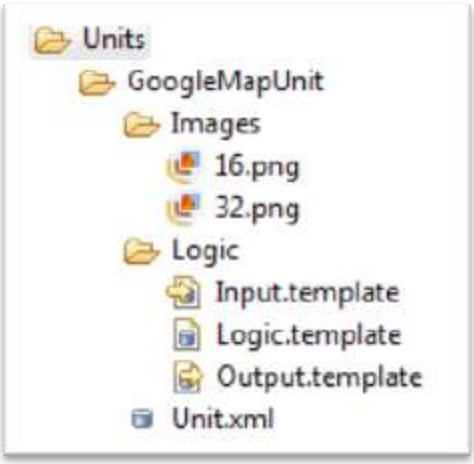

Figura 0-2 – La struttura dei files di una unità personalizzata all'interno dell'albero di WebRatio

Quando un progetto di tipo Unit è aperto in WR tutte le Unità valide presenti in esso sono automaticamente disponibili in ogni progetto WR ed immediatamente utilizzabili.

Emmanuele De Andreis – manudea@duemetri.net

La creazione di una nuova Unità inserisce diversi nuovi file nel progetto, ognuno dei quali ha un preciso significato che è importante comprendere per realizzare delle Unità personalizzate potenti ed efficaci.

Unit.xml

Questo è il file più importante della nuova unità e gestisce tutti i parametri principali. E' un file xml, ma WebRatio mette a disposizione una piacevole interfaccia, suddivisa in diverse sottopagine, che guida nella scelta delle varie opzioni.

Pagina generale (General page)

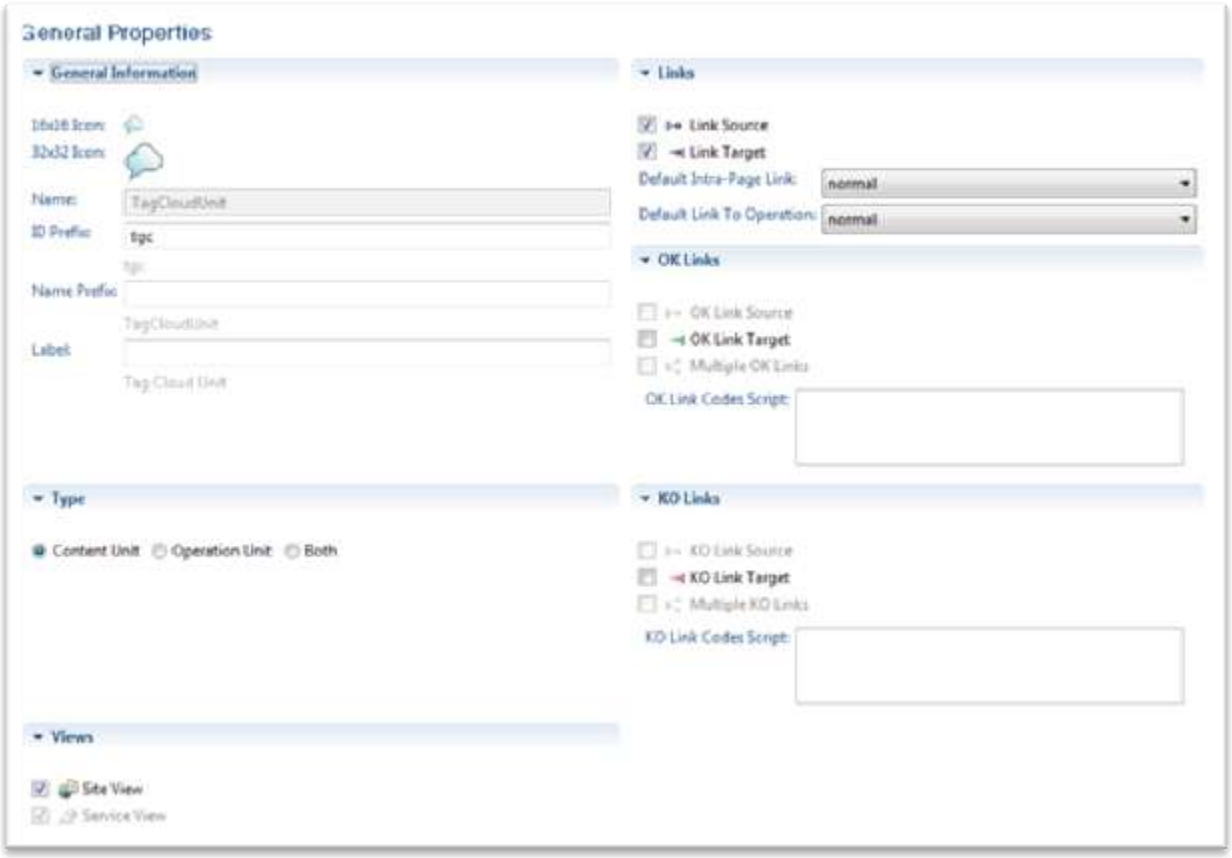

Figura 0-3 – La pagina generale di Unit.xml

La pagina generale ha un duplice scopo: contiene una sezione per impostare le informazioni generali dell'unità come il nome, il tipo, e le viste (Views) dove può essere inserita e specifica se l'unità può essere di origine e di destinazione di uno o più collegamenti.

Le immagini che rappresentano l'unità nelle varie visualizzazioni vanno inserite nella cartella "Images" e devono chiamarsi 16.png (formato 16x16) e 32.png (32x32).

Emmanuele De Andreis – manudea@duemetri.net

Pagina sotto elementi (Sub-elements)

Permette di definire tutte le proprietà che saranno visibili nella finestra delle proprietà dell'unità. Si possono scegliere diverse tipologie di proprietà, dai valori più semplici (testo (string), vero/falso (boolean), numerico intero (integer) o a virgola mobile (float)), quelli che richiamano i moduli HTML (elenco valori (enumeration), password, file) fino a quelli più complessi e tipici di WebRatio che permettono di interagire con le entità ed il database (entità (entity), campi di entità (attribute), relazioni (relationship) e database).

Da questo punto è anche possibile definire sotto unità (che vanno oltre gli scopi di questa trattazione e che quindi non considereremo, fare riferimento alla guida se interessati).

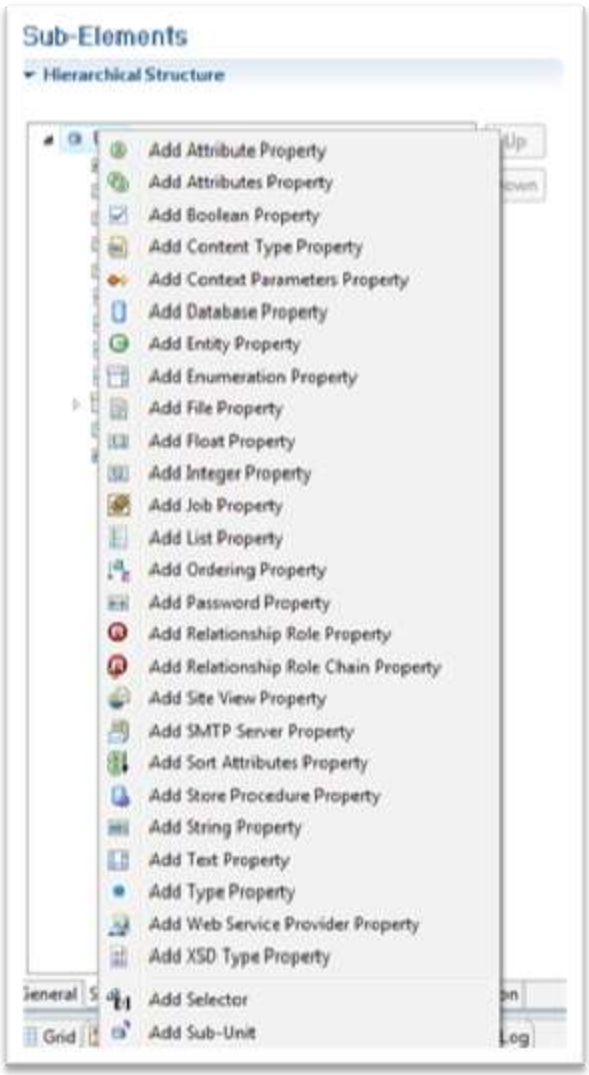

Figura 0-4 – Il menu di inserimento proprietà nella pagina dei sottoelementi.

Va ricordato però che tutti i sottoelementi che sono definiti in questa pagina fanno distinzione tra maiuscole e minuscole.

Emmanuele De Andreis – manudea@duemetri.net

In caso contrario si può ottenere un errore del tipo:

```
com.WebRatio.rtx.RTXException: Error creating service: gmapM2 - WEB-INF/descr/gmapM2.descr
…
Caused by: com.WebRatio.rtx.RTXException: The attribute 'latitude' is missing for the ele-
ment GoogleMapMarkerUnit
     at com.WebRatio.rtx.core.DescriptorHelper.getAttribute(DescriptorHelper.java:209)
```

E' bene pertanto seguire una convenzione che ci aiuti in seguito a non sbagliare riferimenti (ad esempio decidere che tutte le variabili sono scritte in minuscolo).

I sotto-elementi hanno alcune proprietà comuni:

- *Etichetta*: definisce l'etichetta utilizzata per visualizzare le proprietà nella visualizzazione delle proprietà.
- *Ordine in figura*: Se presente permette di visualizzare l'attributo sotto l'icona nel modello WebRatio.

Oltre alle proprietà comuni ogni elemento possiede delle sue proprietà che sono per lo più intuitive (per un elenco completo consultare la guida in linea di WebRatio).

Logica (Logic)

Permette la definizione di script logici. Gli script sono significativi solo per le unità di contenuto, perché gli script permettono di influenzare l'ordine in cui vengono computate le unità sulla pagina e di definire le dipendenze con le altre unità.

Il metodo *computeParameterValue* viene invocato dal runtime di WebRatio in base a come i parametri devono essere propagati all'interno della pagina. La logica con cui i parametri devono essere propagati dipende dal tipo di unità: l'unità stessa dice, attraverso uno dei suoi script di configurazione, di che tipo è:

- *Context-free* (l'unità non ha bisogno di nessun parametro per potersi computare)
- *History* (l'unità mantiene la storia delle scelte fatte dall'utente).
- *Multicondition* (l'unità ha una o più condizioni che le servono per potersi computare)

Se l'unità non ha definito di che tipo è, il runtime non prova nemmeno a propagarne i parametri.

Bisogna aprire il file di configurazione, andare nel tab Logic e aggiungere gli script per definirne il tipo (Context Free, History, Multicondition). Come prima prova è sufficiente aggiungere lo script Context Free e scrivere come valore semplicemente

```
return true
```

A questo punto, dopo aver rigenerato il runtime dovrebbe invocare il metodo computeParameterValue.

Altro valore comune, che lega il contesto ai link presenti in ingresso a runtime, è:

```
return getIncomingLinksAllTypes(unit).empty
```

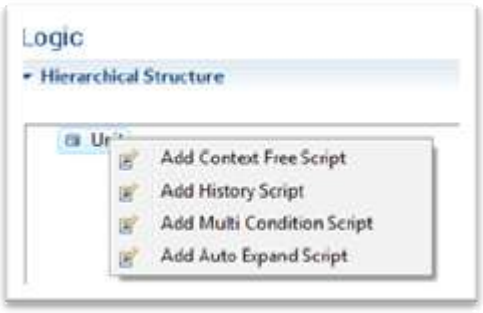

Figura 0-5 – La pagina logica

E' un argomento abbastanza avanzato per cui rimando alla guida in linea per eventuali approfondimenti.

Disposizione (Layout)

Le opzioni in questa pagina permettono di definire il comportamento degli elementi dell'unità ed ha senso solo per unità di tipo contenuto.

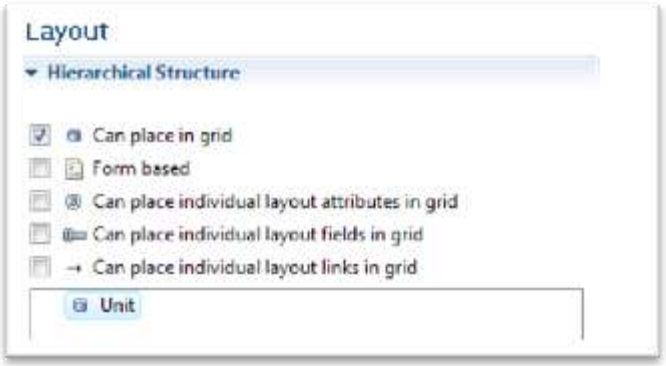

Figura 0-6 – La pagina layout

Le opzioni più importanti qui sono:

- Può essere posizionata sulla griglia (*Can place in grid?*) che decide se l'unità può essere posizionata usando la griglia.
- Basata su modulo (Form based) che regola la possibilità o meno di accettare input da parte dell'utente.

Altre opzioni sono disponibili (consultare la guida in linea a proposito).

Documentazione (Documentation)

Permette di documentare tutte le funzioni che abbiamo deciso di aggiungere alla nostra unità e spiegarne l'utilizzo previsto.

Versione (Version)

Permette di inserire alcune informazioni relative alla versione della unità che stiamo sviluppando.

La cartella logic

L'input e l'output dinamico prodotto dalle istanze delle unità richiede una coppia di templates scritti in Groovy chiamati Input.template e Output.template memorizzati nella sotto cartella Logic della cartella della unità. Lo scopo di questi templates è quello di produrre due frammenti XML che contengono rispettivamente gli elementi <InputParameters> e <OutputParameters>.

Il file Input.template

Il file Input.template specifica i parametri di input.
Ogni parametro di input ha un nome (name), un tipo (tipo) e una etichetta (label).
L'attributo tipo è molto importante perché, se usato correttamente, permette di abilitare la funzione di agganciamento automatico *(automatic coupling)* quando si passano le informazioni su un link in entrata all'unità personalizzata. La logica utilizzata nell'agganciamento automatico è tale da agganciare due parametri se hanno lo stesso tipo di dato, il nome non viene controllato anche se a volte sarebbe utile (se ci sono due parametri con lo stesso tipo di dato vengono collegati automaticamente entrambi al primo dei due anziché ognuno al suo corrispondente).

```
#?delimiters <%,%>,<%=,%>
<%
     setXMLOutput()
     def unitId = unit.valueOf("@id")
%>

<InputParameters>
     <InputParameter name="<%= unitId %>.zoom" type="integer" label="Zoom"/>
     <InputParameter name="<%= unitId %>.latitude" type="double" label="Latitude"/>
     <InputParameter name="<%= unitId %>.longitude" type="double" label="Longitude"/>
</InputParameters>
```

Il file Output.template

Il file di output è assolutamente speculare al file di input, valgono le stesse considerazioni già fatte, cambia solo il nome.

Se il bean ha un metodo corrispondente ad un parametro in output (es: Zoom ha come corrispondenza un getZoom() nel bean) allora il type che avevamo messo in input nell'output non serve.

Inoltre, in generale solamente i parametri di ingresso *(input parameter name)* vanno qualificati con l'id dell'unità per evitare collisione di nomi nella URL.

Figura 0-7 – Il risultato dell'accoppiamento automatico in WebRatio

A titolo di esempio possiamo definire in un'unità il seguente output, che combinerà perfettamente con l'input dell'esempio precedente, risultato che vediamo in figura.

```
#?delimiters <%,%>,<%=,%>
<%
     setXMLOutput()
     def unitId = unit.valueOf("@id")
%>

<OutputParameters>
     <OutputParameter name="<%= unitId %>.zoom" label="Zoom"/>
     <OutputParameter name="<%= unitId %>.latitude" label="Latitude"/>
     <OutputParameter name="<%= unitId %>.longitude" label="Longitude"/>
</OutputParameters>
```

Il risultato dell'accoppiamento automatico in WebRatio si vede dopo aver definito parametri compatibili.

Il file Logic.template

Il file Logic.template specifica le proprietà statiche dell'unità.

Il file contiene un elemento <Descriptor> che descrive il comportamento e l'aspetto delle diverse proprietà. Gli elementi figlio di <Descriptor> contengono i valori delle proprietà disponibili nel quadro proprietà dell'unità.

Tipicamente le proprietà sono già state definite nella pagina dei sottoelementi del file unit.xml, il seguente codice (che viene creato automaticamente quando creiamo la nostra nuova unità) non fa altro che serializzare le proprietà definite, di solito non abbiamo bisogno di scrivere altro.

```
#?delimiters <%,%>,<%=,%>
<Descriptor service="com.WebRatio.units.custom.googlemapunit.GoogleMapUnitService">
     <% printRaw(serializeXML(unit, false)) %>
</Descriptor>
```


Emmanuele De Andreis – manudea@duemetri.net

E' importante notare qui il nome della classe java presente nell'attributo service dell'elemento Descriptor, che vedremo nel paragrafo successivo.

Nel caso fosse necessario è possibile aggiungere elementi personalizzati o con una struttura diversa (vedere la documentazione se si vuole seguire questa strada).

Il file WebModel.template

Il file WebModel.template, è situato nella sottocartella Logic della cartella dell'Unità e permette allo sviluppatore di controllare delle condizioni di errore ed avvisare tramite errori o avvisi in fase di compilazione che l'utilizzo della unità non è corretto o mancano parametri. La definizione di questo file non è obbligatoria, ma può essere molto utile se avete intenzione di sviluppare un'unità personalizzata complessa.

Il codice Java

La creazione di una nuova unità richiede anche la creazione del servizio java che gestisce la logica del componente. In particolare durante il processo di creazione della nuova unità il servizio viene aggiunto nel package di default (com.WebRatio.units.custom.nomeunit se non specificato diversamente). La classe referenziata nel file Logic.template viene creata automaticamente nella directory "src" del progetto Units.

Non tutti i file sono visibili usando la perspective predefinita, per vedere i file contenenti codice java occorre aprire il "Package Explorer" scegliendolo dal menu "Window → Show View". Così facendo sono visibili anche i file della cartella "src" (che contiene il codice sorgente java e javabean).

Il codice che contiene lo stato (Unit Java bean)

Prima di definire il nostro service però è consigliabile creare una classe Java Bean che useremo per gestire tutte le proprietà associate allo stato della nostra unità.

Nel caso di esempio, per semplicità, immaginiamo di avere una sola proprietà di tipo numerico intero chiamata zoom.

Nello stesso package del service creiamo una classe java che ha lo stesso nome della nostra unità, ma con suffisso Bean anziché Service che estende la classe base BasicContentUnitBean.

La classe BasicContentUnitBean è una classe da usare con le unità di tipo contenuto direttamente, oppure estendendola in modo opportuno.

Ha due proprietà:

- **dataSize** (intero) corrispondente alla dimensione dei dati presenti.
- **data** (object) I dati da visualizzare nella unità.

In questa classe aggiungiamo poi una variabile di tipo intero per memorizzare il nostro dato e definiamo i due metodi di get e set.

```
public class GoogleMapUnitBean extends BasicContentUnitBean {

     private int zoom;

     public int getZoom() {
            return zoom;
     }

     public void setZoom(int zoom) {
            this.zoom = zoom;
     }
}
```

Il codice che definisce la logica (Unit Java Service)

Ora possiamo tornare al nostro service.

Nel costruttore della classe leggiamo i valori dal descriptor (quello definito in *Logic.template*).

Il descriptor che andremo a leggere è simile a quello seguente:

```
<GoogleMapUnit xmlns:gr="http://www.WebRatio.com/2006/WebML/Graph" gr:x="0" gr:y="0"
id="gmap1" name="GoogleMapUnit1" zoomFactor="5" customDescriptor="false"/>
```

Si può visualizzare l'XML selezionando un'unità all'interno del modello WR e cliccando sul bottone *'Show XML'* nel pannello XML delle proprietà dell'unità.

Tutti i descrittori xml sono rilasciati insieme all'applicazione generata, nella sottocartella *WEB-INF/descr* nei rispettivi file *<id della unit>.descr*

Lettura dei parametri statici

I parametri statici sono definiti in fase di realizzazione del progetto e sono inseriti direttamente all'interno dell'unità usando la finestra di proprietà.

La lettura dei parametri statici avviene tipicamente nel costruttore della classe service dell'unità.

Per accedere a questi valori per prima cosa accediamo all'elemento base dell'unità (quello che è stato creato dall'istruzione *printRaw(serializeXML(unit, false))* nel *Logic.template*) che si trova nel *descriptor*, passato alla classe service nella variabile *descr*. Per leggere il *descriptor* usiamo le classi di lettura file xml presenti nel package *org.dom4j.Element*

```
     Element currentUnit = descr.element("GoogleMapUnit");
```

Una volta catturato l'elemento possiamo accedere all'attributo e memorizzarlo in una variabile privata della classe, come nel seguente esempio:

```
     Object zoomFactor = DescriptorHelper.getAttribute(currentUnit, "zoomFactor", true,
this);
```

Lettura dei parametri dinamici

I parametri dinamici non sono noti al momento della progettazione del modello, ma sono forniti all'esecuzione da altre unità collegate, che a loro volta possono averli caricati da database, o catturando l'input dell'utente.

Il modello delle unità personalizzate sgancia il comportamento dell'unità dalla sua sorgente dati usando un meccanismo di accoppiamento (*binding*), l'unica cosa che veramente importante è che il tipo dei dati fornito ed atteso corrisponda. In questo modo qualunque unità, almeno in teoria, può agganciarsi a qualunque altra che fornisca dati che lei sia in grado di gestire.

Parametri d'ingresso

Ogni unità può essere sorgente dati (*source*), come definito nel file *Output.template*, oppure ricevere dati (*target*), come definito nel file di *Input.template*.

Per recuperare i dati passati in ingresso è possibile utilizzare, nella classe service, il metodo *asString* dell'oggetto *BeanHelper*, il quale prende in input il nome dell'Input Parameter in questione, recuperato dall'operationContext.

Quindi se per esempio il parametro di ingresso è stato definito in questa forma

```
<InputParameter name="<%= unitId%>.uri" type="string" label="URI"/>
```

allora si può recuperare il valore del parametro in questo modo:

```
String paramString = BeanHelper.asString(operationContext.get(getId() + ".uri"));
```

Il metodo BeanHelper.asString (obj) ritorna il valore stringa di un oggetto. Se l'oggetto di input è una stringa, viene restituito. Se l'oggetto in ingresso è un vettore di stringhe, il primo elemento (se esiste) viene restituito. Se l'oggetto in ingresso è un vettore di oggetti, il metodo *toString()* viene richiamato sul primo elemento (se presente). Infine, il metodo *toString()* viene richiamato sopra l'oggetto d'ingresso. Un valore null viene restituito se l'oggetto di input è null.

Alternativamente si può usare il metodo *BeanHelper.asStringArray(obj)* che ritorna un vettore di stringhe da un oggetto. Se l'oggetto in ingresso è un vettore di stringhe, è restituito un clone. Se l'oggetto in ingresso è una stringa, viene costruito un vettore di stringhe composto da un solo elemento. Se l'oggetto di input è un vettore di oggetti, il metodo *toString()* viene eseguito su ogni elemento per costruire un vettore di stringhe.

```
String[] names = BeanHelper.asStringArray(input)
```

La lettura dei parametri d'ingresso normalmente avviene nel metodo *execute* della classe *service* dell'unità.

Emmanuele De Andreis – manudea@duemetri.net

Parametri di uscita

La lettura dei parametri di uscita avviene nel metodo *computeParameterValue(String outputParamName, Map pageContext, Map sessionContext)* della classe service.

Il metodo è chiamato quando le unità collegate hanno necessità di recuperare un valore dall'unità e la stessa ha dichiarato in che modo chiamarlo negli script *Logic* e *Unit.xml*. Se non è stato dichiarato nulla nella logica, il metodo non sarà chiamato.

Il nome del parametro richiesto è presente come stringa in *outputParamName*, con prefisso dipendente dall'unità, che si può sempre leggere utilizzando la funzione *getId()*.

Il metodo può essere chiamato più volte, tante quante sono i parametri collegati.

Ponendo per esempio che il parametro di output abbia questa forma

```
<OutputParameter name="vote" label="Vote"/>
```

Si controlla che il nome corrisponda e si ritorna il valore corrispondente, o null se non c'è corrispondenza con i parametri noti, in questo modo:

```
public Object computeParameterValue(String outputParamName, Map pageContext, Map session-
Context)
     throws RTXException {
     if(outputParamName.equals(getId() + ".resultObject")) return resultObject;
     return null;
 }
```

Nel codice di esempio abbiamo ipotizzato che esistesse una variabile di classe *resultObject* e che tale variabile contenesse il valore associato al parametro di output omonimo.

Il metodo computeParameterValue(...) viene comunque invocato solo durante la propagazione automatica di pagina (link automatici o transport) . Per link normali entra in gioco la reflection java: se ad esempio l'Output.Template dichiara il parametro di output dal name="vote" dal bean dell'unità viene estratta come conseguenza la proprietà "vote", cioè viene invocato il metodo *getVote()*.

Esecuzione

All'esecuzione del metodo *execute*, che è chiamato quando l'unità viene posizionata sulla pagina, possiamo istanziare la classe *bean* che abbiamo appena creato (che contiene lo stato dell'unità) e valorizzarla con i valori inseriti.

```
public class GoogleMapUnitService extends AbstractService implements
             RTXContentUnitService {

     private int zoomFactor;

     public GoogleMapUnitService(String id, RTXManager mgr, Element descr) throws RTXEx-
ception
     {
             super(id, mgr, descr);

             // Selezioniamo l'elemento che contiene i valori dell'unità corrente
```


Emmanuele De Andreis – manudea@duemetri.net

```
            Element currentUnit = descr.element("GoogleMapUnit");

            // valori predefiniti
            zoomFactor = 5;

            try {
                    zoomFactor = Integer.parseInt(DescriptorHelper.getAttribute(
                                    currentUnit, "zoomFactor", true, this));
            } catch (NumberFormatException ex) { }
    }

    public Object execute(Map pageContext, Map sessionContext)
                    throws RTXException {

            GoogleMapUnitBean map = new GoogleMapUnitBean();
            map.setZoom(zoomFactor);
            return map;
    }
```

La view: Lo stile, ovvero gestire l'output (Layout template)

Lo stile di una unità gestisce il modo con cui l'html viene posizionato sulla pagina e la struttura della relativa jsp.

Gli stili stanno in un progetto di tipo particolare (*Style project*). Se non c'è un progetto di tipo stile nella vostra soluzione ne va creato uno.

Ora dobbiamo creare uno stile per la nostra unità, scegliendo Nuovo → "Layout template" nella cartella Units/Nome unità (se la cartella non esiste va creata).

Quello che viene inserito in questo file verrà renderizzato ed inserito nel file html risultante.

Esiste una sintassi speciale per accedere ai valori definiti nelle classi java che abbiamo creato in precedenza. In particolare per accedere al valore di zoom definito nella classe bean useremo la sintassi:

```
<c:out value="${<wr:UnitId/>.zoom}"/>
```

Questo esempio è una applicazione dalla *JavaServer Pages Standard Tag Library*, una biblioteca standard di funzioni java sotto forma di tag che semplificano lo sviluppo di pagine web **Errore. L'origine riferimento non è stata trovata.**.

JavaServer Pages Standard Tag Library (JSTL) incapsula le funzionalità di base comuni a molte applicazioni JSP, come semplici tag. JSTL ha il supporto per attività comuni quali l'iterazione e istruzioni condizionali, tag per la manipolazione di documenti XML, internazionalizzazione e tag di formattazione di impostazioni locali, tag SQL e strutturali. Introduce anche un nuovo linguaggio di espressione per semplificare lo sviluppo di pagine, e fornisce un'API per gli sviluppatori per lo sviluppo di tag personalizzati conformi alle convenzioni JSTL.

Nell'esempio qui proposto il valore di zoom viene renderizzato in un elemento HTML di tipo <p>

```
#?delimiters [%, %], [%=, %]
[%setHTMLOutput()%]
<p><c:out value="${<wr:UnitId/>.zoom}"/></p>
```

E' possibile definire in una custom unit un layout predefinito in modo che sia automaticamente selezionato ogni volta che l'unità viene posizionata sulla pagina senza richiedere all'utilizzatore di sceglierlo manualmente ogni volta: nel progetto di stile associato al tuo modello c'è un file xml, che si chiama Layout.xml, che permette per ogni unità contenuto di definire i *template* da applicare di default (unit, frame, link, attribute/field) e per ognuno di essi i *layout parameters*.

Per fare in modo che le nuove unità prendano il layout predefinito bisogna fare la stessa cosa anche nelle proprietà progetto WebRatio.

Quando qualcosa va storto

Non è sempre semplice capire cosa non va quando si verificano degli errori.

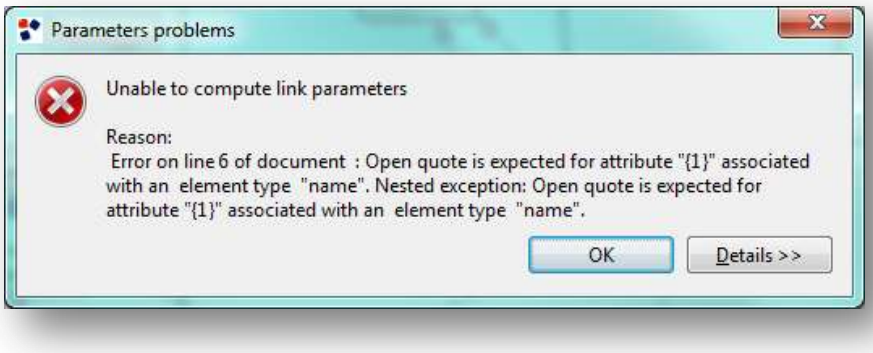

Figura 0-8 – Errore nell'accoppiamento dei parametri.

Un errore possibile che può avvenire quando si tenta di definire l'accoppiamento dei parameri è il seguente, che indica un errore nel file input.template dell'unità personalizzata (nel nostro caso dipendenti dal troppo zelo di word nel convertire le virgolette semplici usate in programmazione con le virgolette tipografiche, più eleganti ma incomprensibili per il parser).

Lo stesso errore può essere ancora più misterioso se avviene quando si cerca di avviare una validazione del modello usando la funzione "Find Model Problems".

Emmanuele De Andreis – manudea@duemetri.net

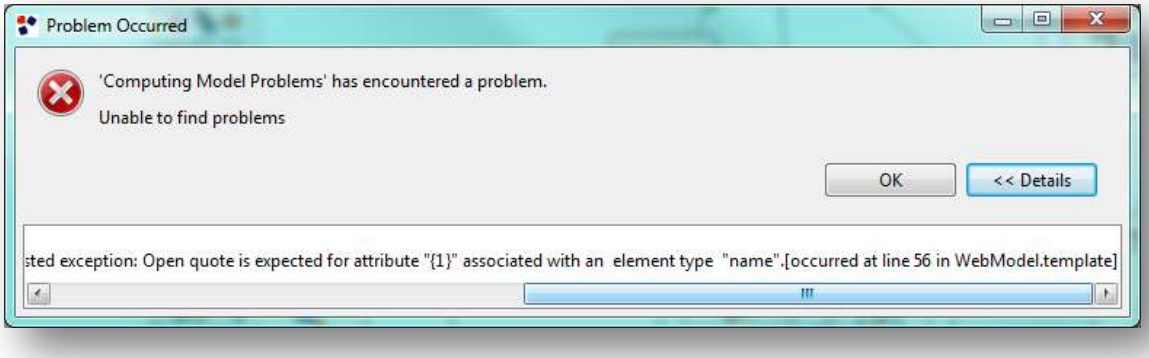

Figura 0-9 – Errore in un file xml

L'errore fa riferimento al Model.Template, che non c'entra nulla, infatti non sempre il messaggio di errore riporta il file corretto.

Un altro errore che si può verificare cercando di aprire una pagina generata è che la pagina stessa non venga trovata.

Figura 0-10 – Errore http 404: pagina non trovata.

Se la pagina non è stata cancellata dal modello è probabile che Tomcat non sia in esecuzione o comunque non sia in grado di rispondere correttamente, a volte capita quando si modificano le unità e Tomcat richiede un continuo ricaricamento del contesto. La prima cosa da provare è rigenerare il progetto completo (*"Generate full web project"*) e riavviare Tomcat.

Un altro errore piuttosto comune è: *"No action config found for the specified url"*, significa che la pagina richiesta non è stata generata.

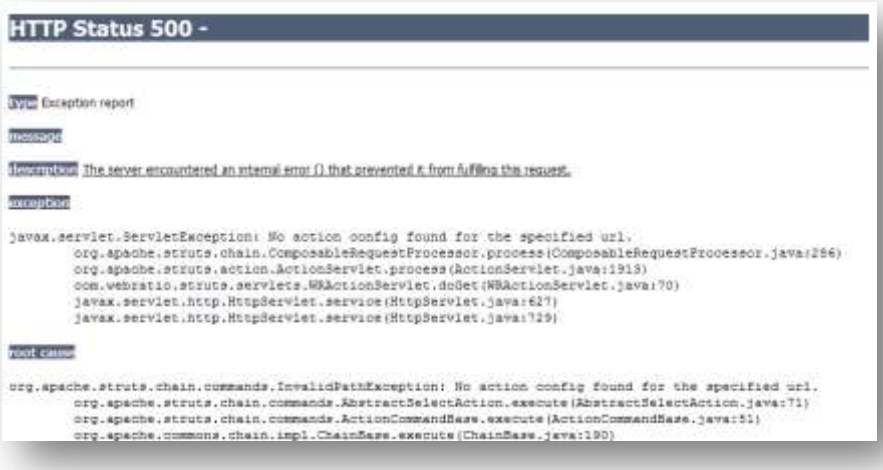

Figura 0-11 – Errore di pagina non generata

Altre volte WebRatio riesce ad essere più esplicito e preciso. Se ci sono errori gravi nella generazione l'applicazione non viene nemmeno avviata.

Figura 0-12 – Errori durante la generazione

Per avere lumi sugli errori di generazione bisogna dare un'occhiata agli errori di generazione usando il comando "Show Details" che troviamo nell'ambiente di WebRatio (oppure visualizzare il sorgente della pagina generata che contiene il dettaglio dello stack con tutto il percorso).

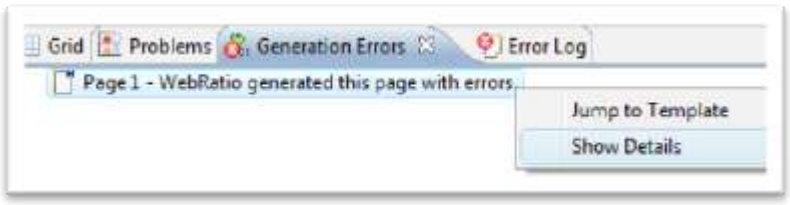

Figura 0-13 – Visualizzazione del dettaglio errori in WebRatio

Emmanuele De Andreis – manudea@duemetri.net

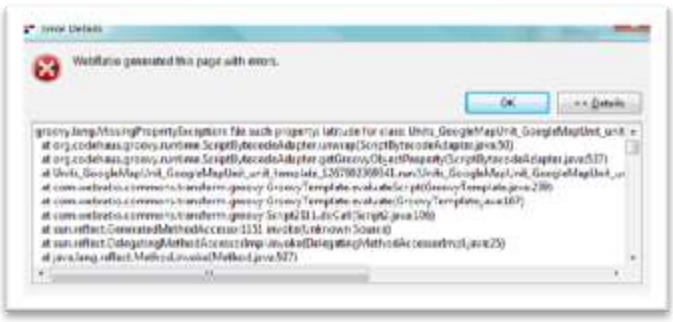

Figura 0-14 – Il dettaglio dell'errore con l'intero stack trace

Quando invece si verifica un errore sulla pagina web è possibile dare una occhiata al codice, che contiene il messaggio di errore completo, commentato e reso invisibile. Il suggerimento arriva dai gruppi di supporto, anche se banale non è documentato.

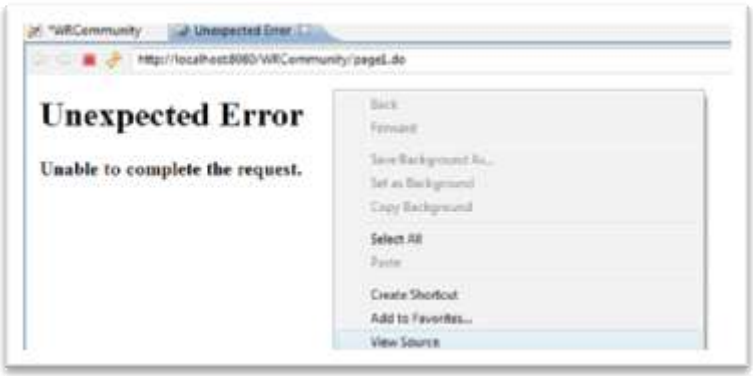

Figura 0-15 – Un errore inaspettato sulla pagina web

Così è più chiaro, avendo a disposizione l'intero stack trace si capisce cosa non va…

Nel caso di un errore di generazione che è possibile imputare ai layout, per trovare la vera causa può essere utile usare la funzione *"Find layout problems"*, in generale, se le unit sono state fatte bene, con le validazioni necessarie, è facile capire il dato mancante che ha causato l'errore di generazione.

Emmanuele De Andreis – manudea@duemetri.net

Debug

Può essere estremamente complicato realizzare una unità personalizzata senza avere la possibilità di fermare ed analizzare l'esecuzione del codice. Fortunatamente il procedimento è semplice e richiede solo alcuni passaggi.

Attivazione tomcat in modalità debug

Per prima cosa bisogna spiegare a Tomcat che deve attivare il debug. Per fare questo bisogna aprire il file catalina.bat (che si trova nella directory di Tomcat: C:\WebRatio\[WebRatio DIR]\Tomcat\bin) ed aggiungere la riga seguente all'inizio del file (subito sotto quella esistente che inizia nello stesso modo: set CATALINA_OPTS=)

```
set CATALINA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8088
```

Bisogna poi far ripartire Tomcat.

All'inizio apparirà una riga del tipo:

Figura 0-16 – Tomcat in debug mode

Attivazione WebRatio in modalità debug

Selezionare in WebRatio il progetto contenente le unità su cui eseguire debug e scegliere "Debug configurations…" dal menu debug e creare una nuova configurazione di debug per una "Remote Java Application".

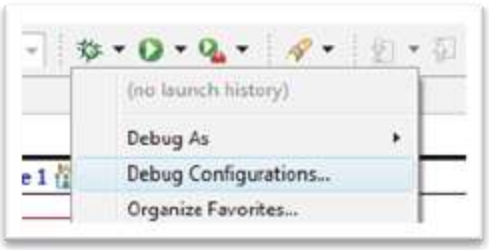

Figura 0-17 – Configurazioni di Debug *… new Remote Java Application*

La porta di debug è la 8088, modalità "Standard (Socket Attach)".

Emmanuele De Andreis – manudea@duemetri.net

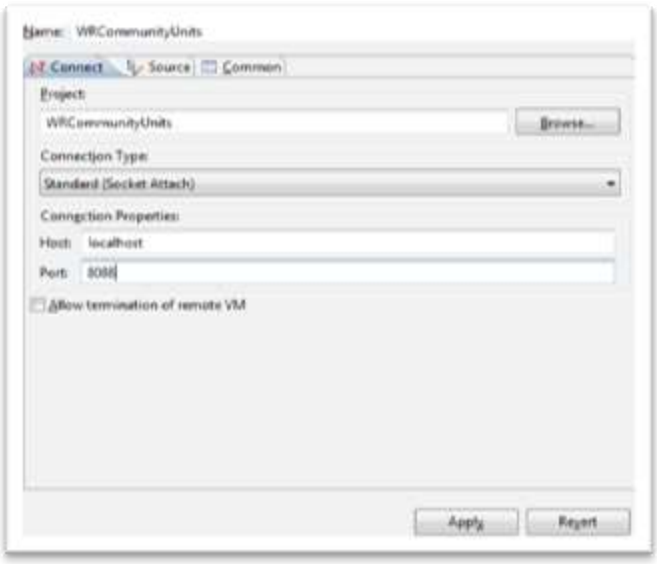

Figura 0-18 – Standard Debug sulla porta 8088

A questo punto basta impostare un breakpoint nel punto dove si desidera che l'esecuzione si interrompa.

Figura 0-19 – Impostazione di un breakpoint

Ora basta generare il progetto e visualizzare una pagina. L'esecuzione si interromperà nel punto desiderato. Per una gestione migliore è consigliabile attivare la *debug perspective* di eclipse, opzione per altro consigliata dallo stesso ide alla prima esecuzione.

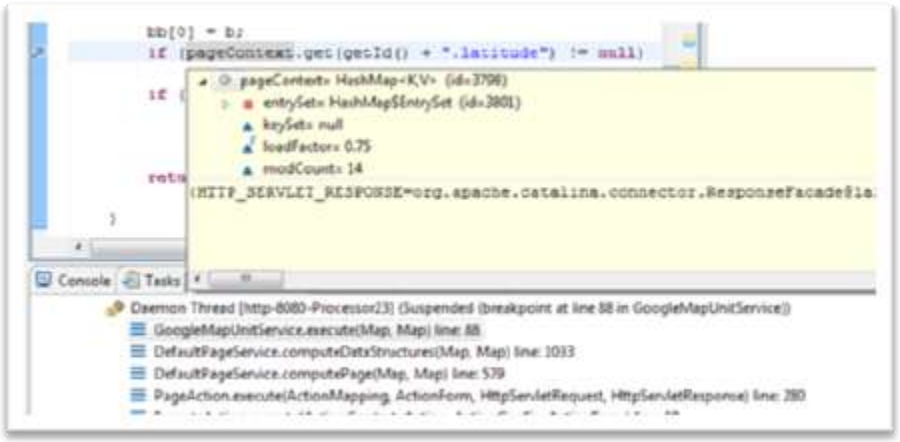

Figura 0-20 – Il debugger in azione

A volte il debugger sembra non funzionare. Per testare se il debugger funziona si può aprire una finestra comando e provare a vedere se con telnet funziona:

```
C:\>telnet localhost 8088
Connecting To localhost...
Could not open connection to the host, on port 8088: Connect failed
```

Se la connessione non funziona la cosa più semplice è provare a riavviare Tomcat, a volte basta.

Se non dovesse bastare occorre rivedere se sono stati eseguiti correttamente i passaggi spiegati in precedenza.

Problemi comuni, soluzioni comuni

Evitare una doppia chiamata al metodo execute

Il numero di volte in cui una unità viene calcolata dipende dall'algoritmo di propagazione. L'algoritmo di propagazione è un algoritmo che opera sul grafo delle unità che compongono una pagina al momento della compilazione. Tale algoritmo dipende dalla struttura della pagina e definisce l'ordine con cui i parametri si devono propagare e l'ordine in cui si devono calcolare le unità. L'algoritmo di propagazione permette di stabilire la logica con cui verrà propagato il contesto all'interno di una pagina nel momento in cui viene invocata. Una unità può comparire più di una volta nell'elenco di propagazione ed è quindi il codice dell'unità stessa che "decide" se ricalcolarsi oppure no in base al fatto che l'unità si sia già calcolata almeno una volta e quindi il suo *bean* sia già presente nel contesto di pagina. È l'algoritmo di propagazione che decide quante volte una unità deve essere computata.

Detto questo, tutte le unità standard implementano nel loro servizio Java una logica per cui, se il *bean* dell'unità è già presente nel contesto di pagina, viene restituito direttamente questo *bean* senza eseguire nuovamente tutto il servizio della unità. Se nelle unità personalizzate create manca l'implementazione di questo meccanismo, è possibile che il *bean* venga processato due volte.

E' possibile implementare questo meccanismo facendo in modo che il metodo *execute*() dell'unità non contenga tutto il codice necessario per calcolare il *bean*, ma richiami una funzione protetta che determina se calcolare il *bean* o ritornare quello già memorizzato nel *page context*. Questa funzione protetta verrà anche utilizzata dal metodo *computeParameterValue()*, che viene invocato ogni volta che l'unità deve propagare del contesto ad altre unità presenti nella pagina.

Qui di seguito un esempio di codice:

```
    public Object computeParameterValue(String outputParamName, Map pageContext, Map ses-
sionContext) throws RTXException {
        Object unitBean = getUnitBean(pageContext, sessionContext);
```

```
        if (unitBean == null) {
            return null;
        }
       ....
   }
```

```
    public Object execute(Map pageContext, Map sessionContext) throws
RTXException {
        return getUnitBean(pageContext, sessionContext);
    }
```

```
    protected Object getUnitBean(Map pageContext, Map sessionContext) throws RTXException
{
        Object unitBean = pageContext.get('_' + getId());
        if (unitBean == null) {
            unitBean = createUnitBean(pageContext, sessionContext);
        } else {
            int dataSize = ((Integer) BeanHelper.getBeanProperty
                           (unitBean, "dataSize", this)).intValue();
            if (dataSize == 0) {
                unitBean = createUnitBean(pageContext, sessionContext);
            }
        }
        pageContext.put('_' + getId(), unitBean);
        return unitBean;
    }
```

Come si può notare, è il metodo *createUnitBean()* che contiene la logica per calcolare il *bean* dell'unità ex novo.

```
protected Object createUnitBean(Map pageContext, Map sessionContext)
throws RTXException {
   ........................
   BasicContentUnitBean bean = new BasicContentUnitBean();
   return bean;
}
```

Utilizzando questo meccanismo, la prima volta che viene computata l'unità, effettivamente viene invocato il metodo *createUnitBean* che calcola il *bean* dell'unità.

Le volte successive che viene richiesto il calcolo dell'unità dall'algoritmo di propagazione, non verrà eseguito questo metodo, ma verrà estratto il *bean* direttamente dal contesto di pagina.

Emmanuele De Andreis

manudea@duemetri.net

Luglio 2010