Abstracting the interface, Part II

Extensions to the basic framework

Martin Gerlach (mgerlac@almaden.ibm.com)
Software Engineer
IBM Almaden Research Center
March 2001

This article, a sequel to the author's December article on Web application front ends, describes extensions to the basic framework for XML data and XSL style sheets. It focuses on the back end this time, covering National Language Support (NLS), enhancements of the view structure, and performance issues. While the first article demonstrated the basic architecture for building Web applications using XML and XSLT, this article shows you how to make the applications ready to go online.

Contents:

Related dW content:

Abstracting the interface, Part I
Improve your XSLT coding five ways

This article is the second in a series on "Abstracting the interface". In order to follow the concepts presented here, you will want to read the first article, Building an adaptable Web app front end with XML and XSL, which discusses the steps necessary to complete the application. Studying the sample application code also will help you grasp the concepts presented here. As in the first installment, this article assumes you are familiar with Web applications based on the HTTP protocol and the HTTP request-response mechanism, and that you have built Web pages using HTML and possibly JavaScript. You should know the Java programming language and how to use Java servlets as access points for Web applications.

What happened so far

In the first article, I introduced WebCal, a simple Web-based calendar application that maintains calendars for multiple users. With this application, I demonstrated how to build a complex Web application that is accessible not only from standard Web browsers, but also from browsers that understand different formats -- for example, WML and VoiceXML for voice interfaces. In that article, I discussed these topics using a three-step process describing:

- A way of structuring the XML data and XSL style sheets
- Server-side XSL transformations -- generating output using XSLT and XPath
- Building HTML forms using XSLT

Where we are going now

I'll also discuss three topics in this article. These are not as tightly connected to each other as the three steps in the first article, and there is no particular order to them. But each of the topics is important in getting the Web application ready to go online. The three topics are:

- National Language Support (NLS): If you are addressing customers in many countries, you will want to provide a multilingual user interface. You want the system to automatically pick up the user's preferred language and display any NLS relevant information in that language.
- Enhancing the application structure: Web applications typically are composed of a number of "views." In the example application WebCal, views show the events of a user, or they display input forms for new events. The login and registration pages are also views. Views can be grouped according to different criteria. In WebCal, the day, week, and month views can be seen as calendar views. They share the same subnavigation -- a link for entering a new event. By identifying ways to group views, developers can avoid redundant coding, thereby easing application development and making it less error prone.

- Performance: You want your Web application to be fast. Users do not want to wait after they follow a link, so you will want to employ caching mechanisms to serve them faster and more reliably.

Let's start with National Language Support. If you are already familiar with NLS, skip to next topic, Enhancing the application structure.

National Language Support (NLS)
The process of giving an application a multilingual UI -- the process of building in National Language Support -- is called "internationalization." Internationalization is a central issue of Java programs, which usually are designed to run anywhere, on any platform, and, most importantly, in any person's language. The Java programming language has a built-in series of internationalization features. You can find articles on internationalization on *developerWorks*; a good overview on the language part of internationalization (NLS) is given in Jared Jackson's article, Harnessing internationilization.

Typically, there are two aspects to NLS:

- Determining the client's preferred language
- Accessing the resources for exactly that language from anywhere in the application

Determining the client's preferred language
In the case of WebCal (and other Web applications), clients can access the system from all over the world. Ideally, we want to present the user interface in the client's preferred language. There are essentially two ways to do this.

The language code (for example, en-us, de) of the client browser can be extracted from the Accept-Language header of the HTTP request, which lists languages accepted by the browser, as shown in Listing 1. The first value in the list is considered the browser's (or the client's) preferred language.

Listing 1: Example for HTTP Accept-Language header

```
Accept-Language: en_us;en_ca;de;
```

In Listing 2, the preferred language is extracted using HttpServletRequest.getHeader("Accept-Language") and basic Java String operations.

Alternatively, you may allow users of your Web app to opt for a language as a user preference. Based on the user's choice, you can then build the language code. All you need to do is create a user-preferences page containing a form that offers all your user preferences. The principles of how this can be done are shown in my first article, under Step 3, "Using forms."

Why do we have to go through the process of analyzing the HTTP headers or user preferences? In internationalized stand-alone Java applications, the language code is queried from the system. When doing the lookup, the `ResourceBundle` class takes care of finding the correct resources for the system language code. In our case, however, the user's language (client language) is entirely independent of the language of the application's system (the server language). We have to work with a fixed language code, representing the user's preferred language.

User prefs vs browser locale
You may use a combination of user preference and browser input by giving the user the choice to use his browser locale as his language or locale preference. This also applies to similar preferences, for example a 12- or 24-hour clock.

Accessing resources for the client's preferred language
Once we have successfully determined the correct language code for the client currently being served, we need a way of looking up translated text using a fixed language code. Listing 3 shows how this can be achieved, using the Locale object locale from Listing 2.

In my first article, I presented an application based on an XML/XSL framework for Web applications. Within this framework we need to differentiate between two places where NLS is needed: Within Java code and in XSL style sheets.

NLS in Java
It is necessary to carry out NLS lookup in Java for any text that is being displayed. In our case, the XML data may contain text to be displayed -- that is, to be copied from the XML to the client format by the XSL transformation. In this case the `ResourceBundle` class (see Listing 3) can be used to look up translated string constants and to include the internationalized strings in the XML data.

NLS in XSL

Parts of the UI may be unique to an output format. For example, we may not want to display the same input field labels within a Web browser and a WAP phone. For the WAP phone we probably want to use very short labels, whereas within a Web browser, labels can be longer and can have features like fly-over help for display. If this is the case, the XSL style sheets for one format may contain text, which may not be part of style sheets for the same view in other formats. In the WebCal XSL style sheets, I used a number of string constants: `Login`, `Username`, `Password`, `Day`, `Week`, `Month`, all the days of the week, and more. In order to internationalize these, we either need to find a way to do a resource bundle lookup in XSL, or we need to do the lookup during the XML generation (that is, in Java) and add all needed translation to the XML document to allow them to be used by the style sheets.

With James Clark's XSL processor XT (see Resources), which I used for WebCal, either approach is possible. XT allows callbacks to static methods of any class in the classpath at the time of XSL processing.

Suppose we have a static lookup method in com.ibm.almaden.webcal.WebCalUtils as shown in Listing 4.

If the language code is available from the XML (that is, if it has been added by the Java code for XML generation), you can call this method from your style sheets. Assuming that the language code has been added as attribute `lang` of the root element `webcal` in the XML data, you can use XSL instructions like the ones highlighted in Listing 5 to do the lookup.

For this code sample to work, the resource bundle used in `nlsLookUp()` (Listing 4) must define the keys SUNDAY, MONDAY, and so on, in each of its language files. It is also possible to use data types other than strings to pass to and from XSL to static Java methods. But to avoid too much conversion overhead, I recommend using strings.

There are some restrictions to the overloadability of methods that are called from XSL. See the XT documentation (in Resources) for details. Using callbacks into Java doesn't have the same performance results as you get doing the lookup during XML generation and including the translated strings in the XML data prior to XSL processing.

More internationalization
Beyond language, there are more issues to NLS and internationalization. If you plan to use time, date, or currency values, you might want to use flexible displays and entry fields.

NLS wrap up

Taking a close look at National Language Support, I explained how to determine the user's preferred language and do NLS lookup for that language in Java as well as in XSL style sheets.

The next section will examine how the application structure can be improved to make your code easier to maintain and extend.

Enhancing the application structure

If you look at the sample WebCal application, you will notice that all calendar views not only share the same main navigation and general links (Home and Logout) but also the same subnavigation bar containing the New Event link. When I described the sample application in my first article, the subnavigation was said to be specific for every single view.

In more complex Web applications, you might want to put views into groups sharing common features -- for example, layout and navigation -- so you do not have to define those again and again for every view, and you can change them by modifying only one piece of code and not the style sheets for every single view in the group. WebCal, the sample application, does not show this, but there are essentially two ways of grouping views:

- By defining groups in the navigation metadata
- By using cascading XSL style sheets

Defining groups in the navigation metadata (WebCal: /web/<format>/navigation.xml)

If you look at the navigation.xml file in web/html/navigation.xml, you'll find the following xml code for the calendar views:

Listing 6: Ungrouped views

```
<actions>
...
  <action name="ShowDay">
    <sub-nav>
      <link type="internal" text="New Event" href="ShowNewEvent">
        <pass-param name="date"/>
      </link>
    </sub-nav>
  </action>
  <action name="ShowWeek">
    <sub-nav>
      <link type="internal" text="New Event" href="ShowNewEvent">
        <pass-param name="date"/>
      </link>
    </sub-nav>
  </action>
  <action name="ShowMonth">
    <sub-nav>
      <link type="internal" text="New Event" href="ShowNewEvent">
        <pass-param name="date"/>
      </link>
    </sub-nav>
  </action>
  ...
</actions>
```

A structure like this can become difficult to maintain: If you wanted to add a new subnavigation link for the three calendar views, you would have to add it to all three `<action>` elements. Wouldn't it be easier to maintain XML through grouped views, as shown in Listing 7?

Listing 7: Grouped views

```
<actions>
  ...
  <group name="calendar">
    <sub-nav>
      <link type="internal" text="New Event" href="ShowNewEvent">
        <pass-param name="date"/>
      </link>
    </sub-nav>
    <action name="ShowDay">
      <sub-nav/> <!-- no view specific sub navigation -->
    </action>
    <action name="ShowWeek">
      <sub-nav/> <!-- no view specific sub navigation -->
    </action>
    <action name="ShowMonth">
      <sub-nav/> <!-- no view specific sub navigation -->
    </action>
```

```
  </group>
  ...
</actions>
```

Doing it this way, you could add your new subnavigation link to the `<sub-nav>` node of the `<group>` node, and it would show up in all the three calendar views. By only having to define the link in one place, the structure gains maintainability. This may not seem to be much of an advantage in this example, but Web applications are typically much more complex than WebCal.

For simplicity, we assume that with grouping, there are no ungrouped views. This means the `<actions>` element in navigation.xml has no `<action>` children. The `<actions>` element has only `<group>` children, which then have at least one `<action>` child (basically, this means there is no `<action>` outside a `<group>` node). Also, the Java code that is responsible for generating the XML for the view needs to include an XML element containing the group of this view (or action) -- for example, `<groupname>calendar</groupname>`, as shown in Listing 8.

Listing 8: View XML

```
<webcal>
  <http-params>
    <param name="action" value="ShowWeek"/>
    <param name="date" value="2001-02-11"/>
  </http-params>
  <groupname>calendar</groupname>
  <user>...</user>
  <navigation> ... </navigation> <!-- groups and actions as shown above -->
  <now date="2000-11-07T15:30"/>
  <week date="2001-02-11" previous="2001-02-04" next="2001-02-18">
    <day date="2001-02-11"/>
    ...
  </week>
</webcal>
```

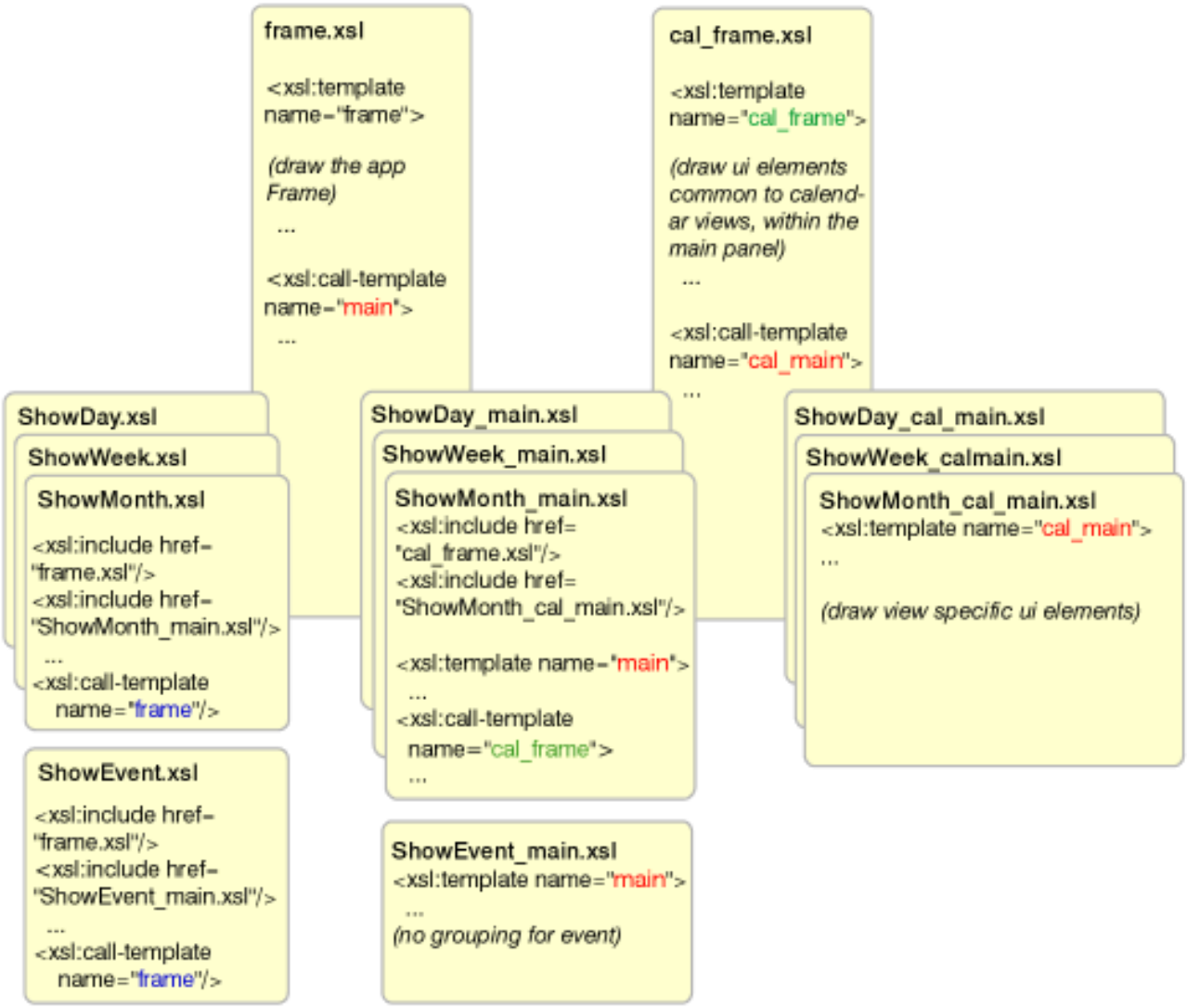
In Listing 26 of the first article, I showed a piece of XSL code that created the subnavigation links for a view. Listing 9 shows the XSL code for generating the subnavigation with view grouping. It shows how to retrieve the current action and group from XML data (as shown in Listing 8). (In this listing, *action* denotes specific actions on the server triggered by the HTTP parameter `action`. Such an action has as a result a view, which is then displayed.)

In WebCal, the subnavigation would be the only navigation criteria that distinguishes views and groups, but more complex applications might have several parameters that can be handled by grouping mechanisms. Another grouping mechanism consists of using cascading XSL style sheets.

Cascading XSL style sheets

In addition to forming groups in the navigation metadata, `<xsl:include>` can be used within the style sheets for the main panel (`WeekViewMain.xsl`, for instance). This way, style sheets for views of a specific group would include XSL templates used by all views throughout the group. This is analogous to many Java methods calling one utility method. The first article described how all views are built from a general frame, defined in `frame.xsl`, and a main panel, which is view specific. In the main panel of views that belong to one group, a group-specific "inner frame" style sheet could be included to generate parts of the response that all views in the group have in common. The inner frame would then at last make the call to the view-specific style sheet, in the same way in which the main template is called from within `frame.xsl`. This cascading mechanism is shown in Figure 1.

Figure 1: Subframes



The cascading mechanism can be implemented down to any desired depth, thus allowing groups within groups.

A recap: Enhancing the application structure
This section demonstrated how you can group the views that your application offers in order to facilitate development and change management. I recommend two ways to achieve this grouping:

- Allow groups to be defined in XML metadata -- for example, as in the navigation information
- Use cascading XSL style sheets

Performance: A look at caching
Now it's time to take a look at the performance of our Web application. There are several ways to improve the performance of a raw application like the WebCal sample application, and one of the most powerful is caching.

The process of generating XML and then transforming it to the client's preferred format using XSLT involves a number of steps. Depending on what the requirements for the system are, there are several starting points for introducing caching. Basically, performance can be gained by caching intermediate results of the transformation, where as a hash key for the cache, a combination of the request URL (that is, the action name and all HTTP parameters), the username, and the desired format can be used.

There are three methods of caching, which can be used in combinations:

- Caching XSL style sheets
- Caching XML
- Caching responses, meaning XML already transformed to the user's preferred format using XSL style sheets

I'll take an in-depth look at all three methods.

Caching XSL style sheets
Looking at the code shown for the XSL transformation (Listing 16) in the first article, you can see that every time the XML response to a request (either generated or retrieved from the cache) is transformed to the client's preferred format, the following two steps are performed before the actual transformation occurs:

- An instance of an XSLProcessor class is created.
- The style sheet needed for the transformation is parsed in by that processor.

The `XSLProcessor` instance can be cached after parsing in the style sheet, and then reused. Because the processor is independent of the requestor and the parameters, the hash key for caching simply can be composed of the value of the `action` HTTP parameter. This parameter determines the view that is to be shown, and the client's preferred format (HTML for instance). This means that after one user has requested a view, the cached processor instance can be used by any user for all other requests of the same view.

The benefit of this approach is that it avoids having to instantiate an `XSLProcessor` and parsing in the style sheet every time a view is requested. That means that one cached processor can be used by all users, which is very efficient.

This approach can be implemented in WebCal in the method `WebCalUtilities.transform(...)`, as shown in Listing 10. It uses the style sheet URI as hash key, which contains action name and format (the URI has the format: `http://localhost/webcal/<format>/<actioname>.xsl`).

Caching XML
WebCal already implements a simple performance tuning: The navigation.xml file or files for each format are read in when the servlet is first loaded and initialized. The resulting navigation XML document fragments are cached by format and are then appended to every view XML that is generated as a response to a client request.

The next step would be to cache the complete XML documents that are created upon requests. To do this, you simply use a hashtable or a similar collection to store the XML documents. By doing this, every time users go back to a page they have already seen, the request URL will look more or less the same -- at least the relevant parameters, user name, action name, desired format, and view specific parameters, will be the same. If the hash key is calculated from those parameters, previously generated versions of the same view can be retrieved easily from the cache. The benefit of this approach is that the back end of the application (a database, EJBs and so on) is accessed less frequently.

This approach can be implemented in WebCal in the method `ShowAction.displayResult()`. I favor the next approach, though, which is similar but more efficient. See listings 11 to 13 for a code example of the next approach.

Caching responses
This approach is a combination of the methods I've just described for caching XML and caching the style sheet. The advantage of caching responses is that it allows responses to be retrieved directly from the server in very little time if the same request has been made before and the result has not been flushed from the cache. The mechanism works similar to the method described for caching XML, only here, the final result of the whole chain of transformation is being cached, using the same hash keys as above. The benefits of this method are that the back end is accessed less frequently *and* that style sheets are not parsed repeatedly.

This approach can be implemented in WebCal in the method `ShowAction.displayResult()`, as shown in Listings 11 to 13.

In order for Listing 11 to work, the modifications shown in Listing 12 must be made to `ShowAction.java` or to its superclass, `Action.java`:

Listing 12: Modified Action class (in Action.java or ShowAction.java)

```
In Action.java:
protected static Hashtable cache = new Hashtable();
...
// wrapper for byte array (inner class)
protected class CachedResponse
{
  public byte[] responseBytes;
  public CachedResponse(byte[] responseBytes)
  {
    this.responseBytes = responseBytes;
  }
}
```

Finally, the method performing the XSL transformation should return the result of the transformation so it can be stored in a cache. I chose `byte[]` (a byte array) as the data type here as this is independent of any specific character encoding. The necessary modifications are shown in Listing 13.

The drawbacks: Flushing the cache and dependencies
Using the cache as a performance enhancement has some drawbacks. Every time a user makes some changes through the Web application interface, XML or response caches must be checked for dependencies. Pages that need to be updated must be deleted from the cache.

The easiest way to deal with this problem is to flush the cache for the user who has made the changes. But then, if the system allows interaction between its users -- for example, data sharing, group calendars, meeting invitations, and schedule changes -- the cache for all users would have to be emptied when just one user changes some data.

To be more efficient, the caches should also store information on dependencies for each page -- the parts of the system that a page depends on. For example, a weekly view in WebCal depends on events that are created, modified, or deleted during the week in which the user is logged in. Dependencies and other additional information, like timestamps to compare the age of the cached data and that of the data in the back end (database), could be stored in the `CachedResponse` class defined in Listing 12.

A positive note: The XSL processor cache does not need to be flushed; XSL processors with associated style sheets can be used until the style sheets change, which typically does not occur at run time.

Beyond caching
The information in the cache can be used to analyze user behavior, such as typical actions performed after login. Using this information, pages could be precomputed and offered to the user in dynamic links, or the precomputed pages can simply be stored in the cache where they can be quickly accessed upon a client request.

Performance: Lessons learned
In this section I showed you how your application's performance can be enhanced through different types of caching: caching XSL processor instances, caching XML, and caching actual results of XSL transformations. The contents of the cache can be used to compute statistics, which in turn can be used to precompute pages that users are likely to request during their sessions.

Conclusion
In my first article, I demonstrated the basics of using XML and XSL transformations to create extensible Web applications. This article followed up on the basics by showing you how to add some advanced features.

I looked at:

- The use of National Language Support to enable global accessibility in multiple languages
- Enhancing the application structure to facilitate maintenance, change management, and further development
- How caching can lead to better performance.

All of these are important elements in giving Web applications a professional touch and in making them ready to go online.

Resources

- Read my first *developerWorks* article on Abstracting the interface: Building an adaptable Web app front end with XML and XSL.
- Download the current and previous Java Development Kits
- Download the Java Servlet Development Kit (JSDK).
- If you want to use WebSphere as a servlet engine, visit IBM's WebSphere page. You can also review the servlet tutorial from WebSphere.
- Download XML4J from IBM *alphaWorks*.
- Learn more about parsing XML files and building XML documents in the *developerWorks*' tutorial XML programming in Java.
- Download XT from James Clark to use for XSL transformations.
- Read the spec for XSLT, the W3C Recommendation for XSLT, edited by James Clark.
- New to XSL? Try the *developerWorks* introduction to XSL.
- For an excellent crash course on XSLT/XPath, see Chapter 14 of *The XML Bible.*
- Try another XSL processor that is fairly easy to use: LotusXSL, also from IBM *alphaWorks*.
- For a good overview on the language part of internationalization, see Jared Jackson's article on NLS in Harnessing Internationalization.
- Benoît Marchal's article on XSLT, Improve your XSLT coding five ways, shows how to integrate client side Javascript and Cascading Style Sheets (CSS).

Download WebCal, the sample application (Windows .zip)

About the author

Martin Gerlach is working as a post grad supplemental in the Computer Science Department of the IBM Almaden Research Center. He holds a German diploma degree in computer science from the University of Applied Sciences of Hamburg. You can contact Martin at mgerlac@almaden.ibm.com.