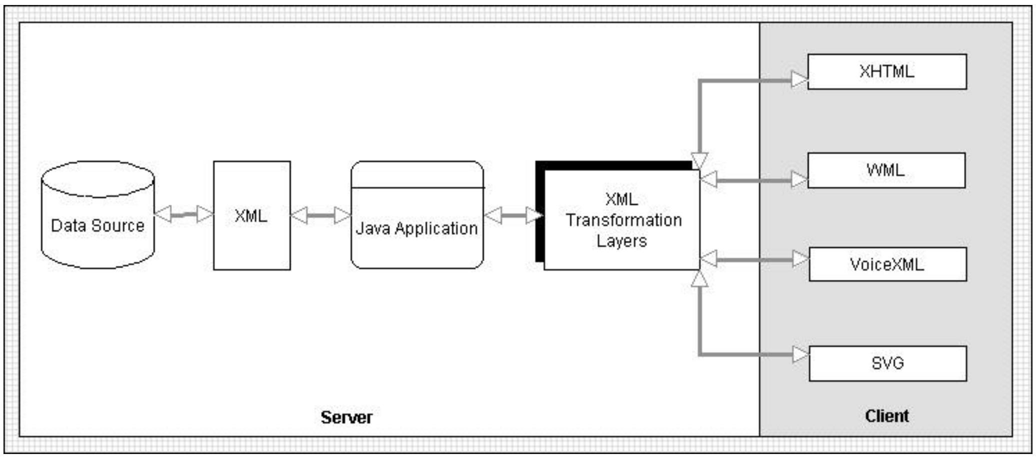
WML Applications using JavaServer Pages (JSP)

Alexander Nakhimovsky, Colgate University

At risk of going over familiar territory, I would like to start this paper by establishing a common background. Here are some facts that I'm assuming you know:

- XHTML is an XML application. So is WML (and the many other MLs!)
- The Web is going to use XHTML and other XML applications. New "HTML" pages should use XHTML
- XML is also being used increasingly for data storage and data interchange
- The simplified, overall picture of a web application looks like this:



It is a Good Thing to commit to a specific presentation language (WML, XHTML) as late as possible — that is, within the XML transformation layer. This presentation is about how to use JSPs in the transformation layer, and how to structure the overall framework so that only the transformation layer need be changed to retarget your application from desktop browser to WAP-enabled phone, to whatever other device/application. Among other things, this design will keep most of your software investment insulated from future changes to WML and WAP, and from device-type proliferation.

JSP is not the only technology that can be used in the XML transformation layer. In addition to direct competitors like ASP and ColdFusion, there is a language called XSLT (eXtensible Stylesheet Language: Transformations) that's specifically designed for transforming one XML document into another. It will be interesting to watch how the two technologies share this space, and we'll look at an alternative XSLT version of our example in this paper too.

Setting Things Up

Assuming Java middleware, the software components we need to perform these operations are:

- Databases: any database that has a JDBC driver. For Access, use http://www.j-netdirect.com/, a solid driver that's free for two connections. There is also a JDBC-ODBC bridge that comes with the JDK; you can use any database that has an ODBC driver with it, but not in production environments. For MySQL, use the driver from www.worldserver.com/ mm.mysql/

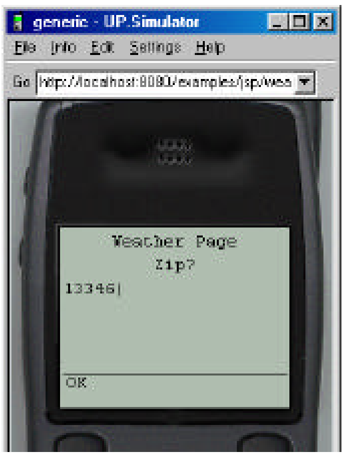
- JDBC: part of Java 2 Standard Edition (J2SE), JDK 1.2.2
- XML Java API: any recent parser from Sun, IBM, Oracle or Microsoft. We use Sun's JAXP (Java API for XML Parsing) reference implementation, from java.sun.com/xml.
- Servlet/JSP engine: the most common combinations are IIS + JRun (www.allaire.com) and Apache + Tomcat. You can also use the reference implementation, Tomcat, by itself, as it has a built-in web server. This is what we will use.
- Client browser and UP Phone SDK

To use XSLT, you will need an XSLT processor implemented as a servlet (for example, James Clark's xt, Michael Kay's Saxon, or Apache Xalan). We'll use xt.

> *James Clark announced recently on xsl-dev that he is unlikely to continue supporting xt, but there is an Open Source group that has declared its willingness to take over. Michael Kay is actively supporting Saxon, and is also the author of* XSLT Programmer's Reference *(Wrox Press).*

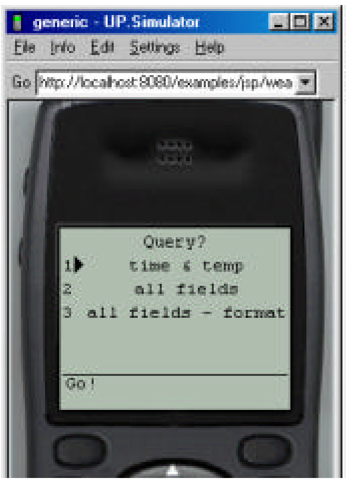
The Look of WAP

This is how our first example looks in the UP.Simulator after entering the URL and the first input:

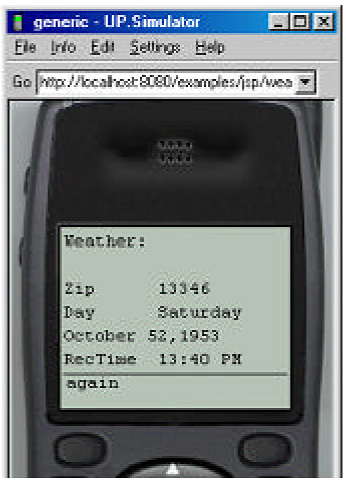

generic - UP.Simulator
File Info Edit Settings Help
Go http://localhost:8080/examples/jsp/wea
OK

```
Weather Page
Zip?
13346
```

Next, clicking on the "OK" softkey, then clicking the "2" number key to pick option #2 ("all fields") gives the following sequence of events:

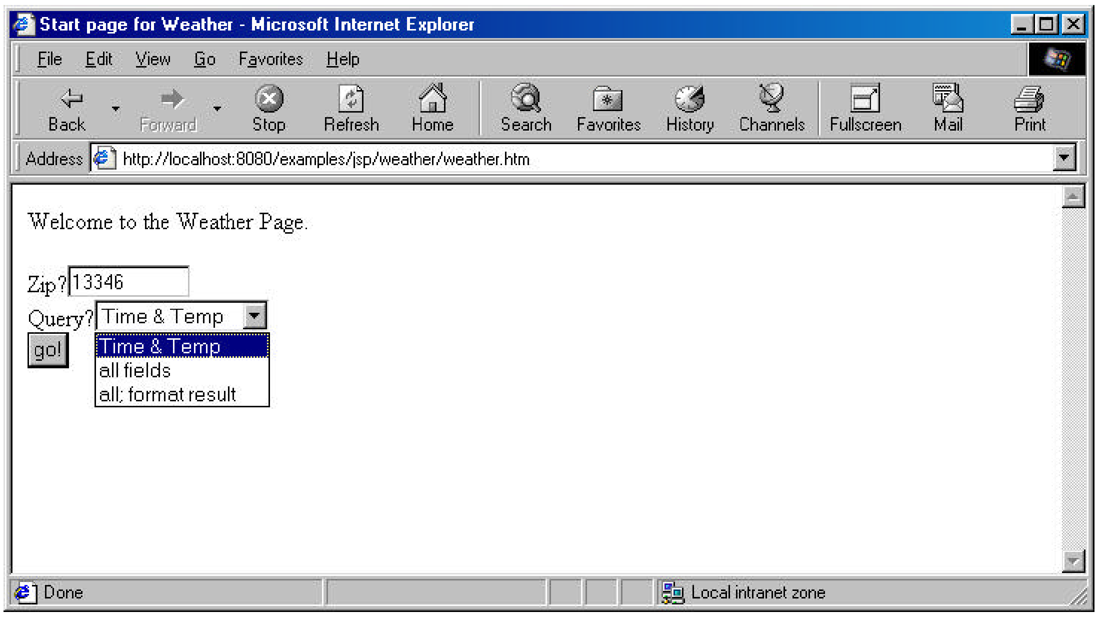

generic - UP.Simulator
File Info Edit Settings Help
Go http://localhost:8080/examples/jsp/wea
Go!

```
Query?
1    time & temp
2    all fields
3 all fields - format
```


generic - UP.Simulator
File Info Edit Settings Help
Go http://localhost:8080/examples/jsp/wea
again

```
Weather:

Zip       13346
Day       Saturday
October  52,1953
RecTime   13:40 PM
```

The same database can be accessed in a desktop browser. Here's a very pedestrian web page, showing the same events as the first two UP.Simulator pictures:


Start page for Weather - Microsoft Internet Explorer

Address http://localhost:8080/examples/jsp/weather/weather.htm

Welcome to the Weather Page.

Zip? 13346

Query? Time & Temp

go!

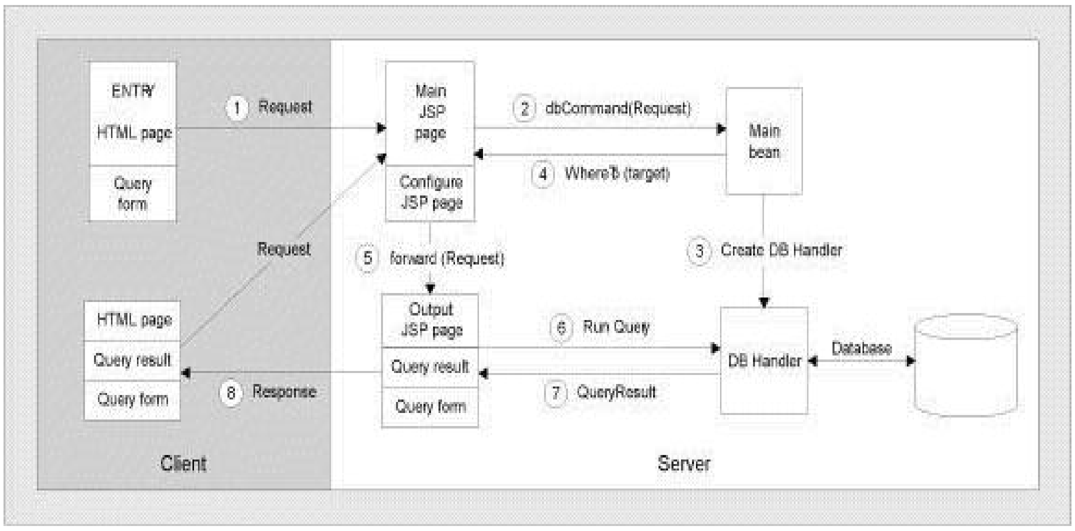
Time & Temp
all fields
all; format result

Done — Local intranet zone

The web and WAP applications both use the same `weather.jsp` page, which resides in the same directory as a file called `configure.jsp`. All the Java code is the same, and it accesses the same database table. It is a plausible strategy to develop your application first for a web client (for ease of debugging) then re-target to WAP.

An Architecture for a Web / WAP JSP Application

The structure of our application looks like this:



This design allows for a continuing conversation between the client, the main JSP, the main bean, and output JSPs. The idea is that a JSP functioning as an output template contains a form (or a <go> element) whose `action` attribute is the URL of the main JSP. The main `weather.jsp` page acts as a switchboard to which all requests are sent, and which forwards them to the right output JSP.

The main JSP interacts with the main bean using two string variables: `beanCmd` and `jspCmd`. `beanCmd` determines what kind of action the bean executes, while `jspCmd` determines the output template that the main JSP selects for sending back the response. Most of those output template pages contain an XHTML form (or a WML

<p> element) with appropriate elements for input. The `action` attribute of the form (or the `href` attribute of the WML <go> element) is the URL of the main JSP.

I will work through this diagram, selectively looking at components. The goal is to learn enough JSP and JDBC to be able to use the framework and develop your own applications of similar structure. We'll start with the WML entry page, `weather.wml`.

The WML Entry Page

Here's the WML code for the first two microbrowser screens we saw, a WML deck consisting of a single card:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//PHONE.COM//DTD WML 1.1//EN"
                     "http://www.phone.com/dtd/wml11.dtd">
<wml>
   <card id="start" title="Weather">

      <do type="accept" label="Go!" >
         <go href="weather.jsp"><!-- submitted to a JSP -->
            <postfield name="query" value="$query" />
            <postfield name="QP1" value="$QP1" />
            <postfield name="target" value="wml-$(query)" />
         </go>
      </do>
      <p align="center">Weather Page</p>
      <p>
         Zip?
         <input name="QP1" format="*N" maxlength="10" value="" /><br />
         Query?
         <select name="query">
            <option value="TimeTemp" > time &amp; temp </option>
            <option value="AllTable" > all fields </option>
            <option value="AllText" > all fields format</option>
         </select>
      </p>
   </card>
</wml>
```

Here's the WML code for the third screen we saw above:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//PHONE.COM//DTD WML 1.1//EN"
                     "http://www.phone.com/dtd/wml11.dtd">
<wml>
   <head>
      <meta http-equiv="Cache-Control" content="no-cache" forua="true"/>
   </head>

   <card id="output" title="AllTable">

      <do type="accept" label="again" >
         <go href="#askCard" />
      </do>
      <p>Weather:</p>
      <p>
         <table columns="2">

            <tr>
               <td>Zip</td><td>13346</td>
            </tr>

            <tr>
               <td>Day</td><td>03/12/2000</td>
            </tr>

            <tr>
               <td>RecTime</td><td>1:40 PM </td>
            </tr>

            <tr>
               <td>Temp</td><td>37</td>
            </tr>
            <tr>
               <td>DayLo</td><td>35 </td>
            </tr>

            <tr>
               <td>DayHi</td><td>46 </td>
            </tr>

            <tr>
               <td>Precip</td><td>60</td>
            </tr>

            <tr>
               <td>Warn</td><td>Foggy, probable rain; caution in driving</td>
            </tr>

            <tr>
               <td>Tomorrow</td><td>We won't have any weather tomorrow</td>
            </tr>

            <tr>
               <td>NextDay</td>
               <td>There will be weather nearby, but not on you.</td>
            </tr>
         </table>
      </p>
   </card>

   <card id="askCard" title="Weather">
      <do type="accept" label="Go!" >
         <go href="weather.jsp">
            <postfield name="query" value="$query" />
            <postfield name="QP1" value="$QP1" />
            <postfield name="target" value="wml-$(query)" />
         </go>
      </do>
      <p align="center">Weather Page</p>
      <p>
         Zip?  <input name="QP1" format="*N" maxlength="10" value="" />
         <br />
         Query?
         <select name="query">
            <option value="TimeTemp" > time &amp; temp </option>
            <option value="AllTable" > all fields </option>
            <option value="AllText" > all fields-format </option>
         </select>
      </p>
   </card>
</wml>
```

Introduction to JSPs

JSPs belong in the 'CGI space' — that part of the system where the web server connects to the middle tier, passing in the client request and getting back the response. JSPs are built on top of Java servlets, which provide a Java framework for dynamic page generation in response to an HTTP request. Servlets require programming skills, but a JSP is simply a template for page output, whether that's HTML, or XML, or some other file format. Segments of Java code are used to add data dynamically to the page before it is returned to the client.

JSPs are similar to ASPs (Active Server Pages) in more than just name, but they use Java rather than a scripting language. If you're familiar with ASPs, you'll find JSPs very easy to get used to, but some people argue that JSPs are a superior technology because:

- Java is more powerful than the scripting languages in ASP: it's multi-threaded, network-savvy, and security conscious
- On first use, the JSP is compiled into a Java servlet class, and after that the compiled code is used every time the page is invoked. This makes JSPs more efficient than ASPs that use a scripting language

We don't aim to proselytize either technology: we simply prefer JSPs because we know Java much better than Visual Basic.

What makes up a JavaServer Page?

A JSP is an HTML or XML page that may contain **JSP elements**. An HTML page with no JSP elements is also a legitimate JSP (just change the extension from `.html` to `.jsp`) and will be displayed normally, after processing.

In processing a page, the JSP processor leaves HTML/XML material untouched, but acts on JSP elements. JSP elements can contain Java code (either Java expressions or Java statements). **Java expressions** are evaluated, and the value is inserted into the HTML/XML page in their place. **Java statements** can control page content: a stretch of HTML/XML material can be repeated if it is inside a Java loop, or included conditionally if it is placed inside a Java conditional statement.

JSP elements are delimited by **JSP tags**. All of these tags have standard XML syntax, within the `jsp:` namespace, but some also have a JSP-specific syntax that is different from XML.

> *In version 1.0, all non-XML tags have equivalent XML tags defined in the JSP DTD, so that a JSP* can *be valid XML. With version 1.1 (which is the current version at the time of writing), JSP processors are required to accept and validate JSPs in purely XML syntax. In effect, JSP 1.1 processors contain a validating XML parser.*

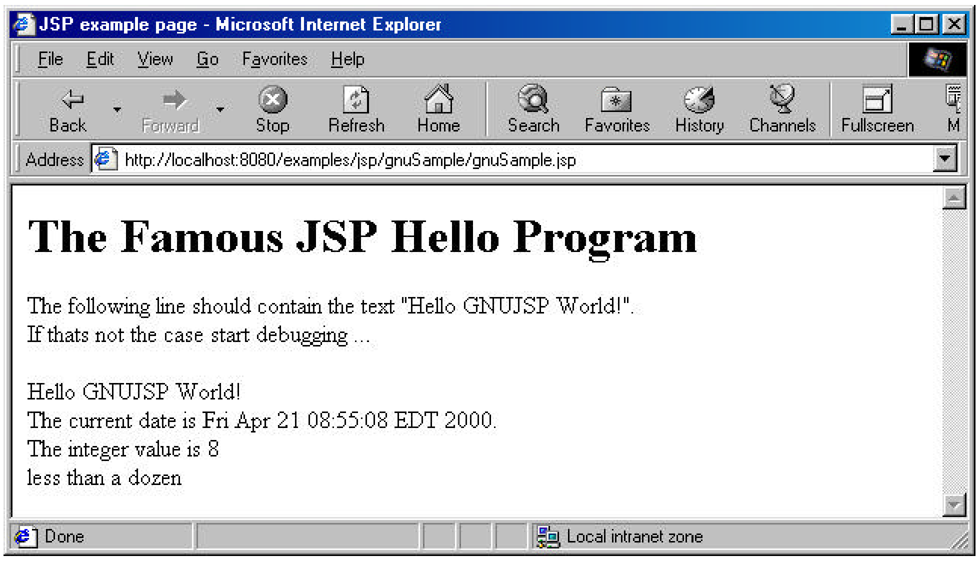
The purpose of JSP tags is to include Java code in the page, and to perform servlet-related tasks: request processing and forwarding, session maintenance, and communication with business objects. Describing these tasks within elements with XML syntax can be awkward because Java code often contains characters that have to be escaped in XML (such as <); as a result, you have to use CDATA sections a lot. In manually composed JSPs, it is common to use non-XML tags, but XML tags are expected to be very useful as JSP editors like JRun 3.0 Studio become available.

What Do They Look Like?

Here's a short example, `sample.jsp`. You can use standard XML comments within a JSP document:

```
<html>
   <head>
      <title>JSP example page</title>
      <%! int i=5,j=2; %>                       <!-- a declaration -->
      <%@ page import="java.util.Date" %>       <!-- a directive -->
   </head>

   <body>
      <h1>The Famous JSP Hello Program</h1>
      <% String s = "GNU" + "JSP"; %>           <!-- a code fragment -->

      The following line should contain the text "Hello GNUJSP World!".
      If that's not the case, start debugging...

      <p>Hello <%= s %> World!<br>              <!-- an expression -->

      The current date is <%= new java.util.Date() %>.<br>
      The integer value is <%= ++i+j %>         <!-- another expression -->

      <% if(i<12){ %>                           <!-- code fragment -->
      <br>less than a dozen                     <!-- template data -->
      <% }else %>                               <!-- code fragment -->
      <br>a dozen or more                       <!-- template data -->
      <% ; // end if %>                         <!-- code fragment -->
   </body>
</html>
```

If you have this document served to you by a JSP-enabled server (a server that has a **JSP engine**), this is what you will see:


JSP example page - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen M
Address http://localhost:8080/examples/jsp/gnuSample/gnuSample.jsp
Done
Local intranet zone

The Famous JSP Hello Program

The following line should contain the text "Hello GNUJSP World!".
If thats not the case start debugging ...

Hello GNUJSP World!
The current date is Fri Apr 21 08:55:08 EDT 2000.
The integer value is 8
less than a dozen

The HTML source for this page is shown below. Notice how all the JSP elements have disappeared, but the comments have remained:

```
<html>
   <head>
      <title>JSP example page</title>
      <!-- a declaration -->
      <!-- a directive -->
   </head>

   <body>
      <h1>The Famous JSP Hello Program</h1>
      <!-- a code fragment -->

      The following line should contain the text "Hello GNUJSP World!".
      <br>If thats not the case start debugging ...
      <p>Hello GNUJSP World!<br>                <!-- an expression -->
      The current date is Fri Apr 07 08:59:07 EDT 2000.<br>
      The integer value is 8                    <!-- another expression -->

      <!-- code fragment -->
      <br>less than a dozen                     <!-- template data -->
      <!-- code fragment -->
   </body>
</html>
```

Click the Reload/Refresh button several times, and you will see the integer value (and eventually the message that follows it) change.

An Overview of JSP

JSP syntax is quite straightforward and compact: it all fits on a double-page syntax card available from Sun (java.sun.com/products/jsp/syntax.pdf). A JSP consists of template data and JSP elements, which fall into the following groups: **directives**, **scripting elements**, **comments**, and **actions**. Scripting elements are further subdivided into **declarations**, **expressions**, and **code fragments** (or **scriptlets**). We'll see their syntax and describe their meaning shortly.

The first three groups have always been part of JSP, and they have non-XML syntax as well as alternative XML+Namespace syntax. The actions group is more recent, and uses only the XML+Namespace syntax.

Non-XML Elements

Non-XML syntax of directives, scripting elements and comments is summarized in the table below. You have seen all of them used in our simple example:

Type of element	**Syntax description**	**Example**
Directives	`<%@ directive %>`	`<%@ page language="java" %>`
Declarations	`<%! declarations %>`	`<%! int i=0, j=5; %>`
Expressions	`<%= expression %>`	`<%= i+7 %>`
Code fragments	`<% code fragment %>`	`<% if(i < j-4 { %>`
Comments	`<%-- comment --%>`	`<%-- not for the client --%>`

What they all Mean

- Directives are addressed to the JSP engine. They do not produce any output. You will see two directives in our output pages: page and `include`. `include` does exactly what its name suggests: it is used to include files into a page. The page directive is for setting the global properties of the page, such as language and content type.
- Within scripting elements, declarations are exactly that: Java declarations and (perhaps) initializations. Declarations do not produce any output.
- Expressions are Java expressions; they are evaluated and their values are inserted into the output stream.
- Code fragments are stretches of Java code. They don't have to be complete statements or valid expressions.
- JSP comments are not sent to the client; they are strictly for documenting code.
- You can also use standard XML comments in a JSP, and they will be treated like regular XML comments. You can even include non-XML JSP content in XML-style comments, and it will be treated as part of the comment — that is, it will be ignored.

Action Elements

Action elements fall into two groups:

- Actions having to do with JavaBeans: useBean, `getProperty`, `setProperty`. Beans are Java classes that conform to some simple patterns: using public get/set methods to access properties, providing a public, no-arguments constructor, and being serializable.
- `include` and `forward` actions

All action tags appear with the `jsp:` namespace prefix; here's an example of using them:

```
<jsp:useBean id="mbean" scope="session" class="weather.MainBean" />
```

Beginning with JSP 1.1, it is possible to define custom actions (using the `jsp:taglib` element). This has the potential greatly to extend the application of JSP and the reuse of common functionality, allowing non-programmers to create JSPs by simplifying the architecture of JSP-based applications.

JSPs in the Application

Our application uses the main JSP page for overall control, `configure.jsp` to configure the system, and several JSP pages for output. I'll show them in that order.

The Main Page, weather.jsp

This page never sees the light of day (in the browser), so it has only JSP elements: two directives, a useBean action, and a piece of Java code. The code carries out the dialog between the main page and the main bean described in the diagram: the main page sends the request object to the main bean for analysis; the main bean tells it to which output page to forward the request. Here's the code:

```
<%@ page errorPage="errorpage.jsp" %>
<%@ include file="configure.jsp" %>

<jsp:useBean id="qBean" class="MyNa.jspUtil.QBean" scope="session"/>

<%
   qBean.doCommand(request);
   if(true) {
      out.clear();
      pageContext.forward(qBean.whereTo());
      return;
   }
%>
```

configure.jsp

Here is `configure.jsp`. Like `weather.jsp`, it contains nothing but Java code:

```
<%
   // Information for connecting to database
   String[] dbParams = new String[] {
      "sun.jdbc.odbc.JdbcOdbcDriver",
      "jdbc:odbc:WEATHER",
      "userName",
      "defaultPassword"
   };

   // Queries available to the client
   String[][] dbQueries = new String[][] {
      {"AllTable", // All fields formatted as a table
       "SELECT * FROM FORECAST WHERE Zip=?"},

      {"AllText",  // All fields formatted as a paragraph of text
       "SELECT * FROM FORECAST WHERE Zip=?"},

      {"TimeTemp", // Date and temperature only
       "SELECT RecTime,Temp,DayLo,DayHi FROM FORECAST WHERE Zip=?"}
   };

   // Associate each query with an output JSP file
   String[][] responseTargets = new String[][]{
      {"AllTable ", "/jsp/weather/xhtml/AllTable.jsp"},
      {"AllText ", "/jsp/weather/xhtml/AllText.jsp"},
      {"TimeTemp", "/jsp/weather/xhtml/TimeTemp.jsp"}
   };

   // This code is actually run once per session
   qBean.doConfigure(dbParams, dbQueries, responseTargets);
%>
```

As you can see, there are three declaration sections in the file, followed by a single function call. The first section has to do with information that is needed to connect to a database using a JDBC driver — that is, information needed to create a `Connection` object.

The second section has to do with running queries. It consists of name-value pairs, where each value is a string you would use to create a `PreparedStatement` object. These values are stored in a dictionary-like object (a `Hashtable`), indexed by the corresponding names. In order to run a query, the user has only to specify its name (in a `select` element of the form) and provide the values for the parameters of the `PreparedStatement`. Since these parameters are ordered, the form elements for entering query parameters must have such names as `Parameter1`, `Parameter2`, and so on (or, to shorten them for the wireless Web where every byte counts, `QP1`, `QP2`). This is a convention that is needed for our `DBHandler`.

The third section associates output templates with query names. It also consists of name-value pairs, where the names are the names of queries, and values are the names of JSP files to forward the request to.

The configuration page includes a call to `qBean.doConfigure()`. The task of `doConfigure()` is to set up an object of our `DBHandler` class, presented in the next section.

Although `doConfigure()` is called once per request, it will have no effect after the session is first set up and a `DBHandler` object is stored in it. (The method starts with an `if` statement that checks to see whether the `DBHandler` is null.)

> *We could also instantiate and configure the main bean, including a `DBHandler`, from an XML configuration file. This would require more background machinery to explain. The method shown here allows a system administrator to configure the database access and alter the target files without introducing too many new concepts.*

JSPs for Output

There are three output pages corresponding to three queries: `TimeTemp`, `AllTable`, and `AllText`. Here is `AllTable.jsp`:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//PHONE.COM//DTD WML 1.1//EN"
                     "http://www.phone.com/dtd/wml11.dtd">
<wml>

   <%@ page errorPage="../wmlerrorpage.jsp" %>
   <jsp:useBean id="qBean" scope="session" class="MyNa.jspUtil.QBean" />

   <head>
      <meta http-equiv="Cache-Control" content="no-cache" forua="true"/>
   </head>

   <card id="output" title="AllTable">
      <do type="accept" label="again" > <go href="#askCard" /> </do>

      <%
         MyNa.jspUtil.QueryResult qR = qBean.queryResult();
         String[][] rows = (null == qR) ? null : qR.getRows();
         if(null == rows || rows.length < 1) {
      %>
      <p>Sorry, no weather in zip-code.</p>
      <%
         } else { String[] headers = qR.getColumnHeaders();
      %>
      <p>Weather:</p>
      <p><table columns="2">
         <%  // We use only rows[0] to generate a 2-column table of fields
            for(int j = 0; j < headers.length; j++) {
         %>
         <tr>
            <td><%= headers[j] %></td><td><%= rows[0][j] %></td>
         </tr>
         <%   }   %>
         </table>
      </p>
      <% } %>
   </card>

   <card id="askCard" title="Weather">
      <do type="accept" label="Go!" >
         <go href="weather.jsp">
            <postfield name="query" value="$query" />
            <postfield name="QP1" value="$QP1" />
            <postfield name="target" value="wml-$(query)" />
         </go>
      </do>
      <p align="center">Weather Page</p>
      <p>
         Zip?
         <input name="QP1" format="*N" maxlength="10" value="" /><br />
         Query?
         <select name="query">
            <option value="TimeTemp" > time &amp; temp </option>
            <option value="AllTable" > all fields </option>
            <option value="AllText" > all fields-format </option>
         </select>
      </p>
   </card>
</wml>
```

Notice how this page gets the result set from the `DBHandler` as a string matrix and dumps it directly to an output table, without any regard for its size. This is OK for a web application, but a risky thing to do in a WAP application. It is for this reason that we also provide an `AllText` query, in which the output is crafted by hand and so can be controlled more precisely. Here are the cards for `AllText.jsp`:

```
<card id="output" title="TimeTemp">
   <do type="accept" label="again">
      <go href="#ask" />
   </do>
   <do type="accept" label="details" >
      <go href="#details" />
   </do>
   <% // Zip,Day,RecTime,Temp,DayLo,DayHi,Precip%,Warn,Tomorrow,NextDay

      MyNa.jspUtil.QueryResult qR = qBean.queryResult();
      String[][]rows = (null == qR) ? null : qR.getRows();
      if(rows == null || rows.length < 1){
      // title row should be always there
   %>
   <p>Sorry, no weather in your zip </p>
</card>

<% } else {
   Dict D = new Dict();
   D.setDef(qR.getColumnHeaders(), rows[0]);
   D.setOutLimit(300);
%>

   <p>
      zip: <%= D.getDef("zip") %> <br/>
      temp: <%= D.getDef("temp") %><br/>
      at <%= D.getDef("RECTIME") %> <br/>
      day's lo: <%= D.getDef("daylo") %> <br/>
      day's hi: <%= D.getDef("dayhi") %>
   </p>
</card>
<card id="details" title="TimeTemp">
   <do type="accept" label="again" >
      <go href="#ask" />
   </do>
   <do type="accept" label="more" >
     <go href="#moreDetails" />
   </do>
   <p>
      day: <%=D.getDef("day") %><br/>
      precip: <%=D.getDef("precip") %>
   </p>
</card>

<card id="moreDetails" title="TimeTemp">
   <do type="accept" label="again" > <go href="#ask" /> </do>
   <do type="accept" label="basics" > <go href="#output" /> </do>
   <p>
      <%  String W = D.getDef("warn");
         if (W.length() > 0) { %>
      Warning: <%= W %> <br/>
      <%  } %>
     Tomorrow: <%= D.getDef("tomorrow") %> <br/>
     NextDay: <%= D.getDef("nextday") %>
   </p>
</card>

<% } %>
<!-- the remaining ask card is the same -->
```

This time, we control the amount of output by means of the `setOutLimit()` method, and we provide three cards of increasing level of detail. Something to be aware of is that the amount of output is controlled on the server, in characters. Testing is needed to see how this limit translates into limits on the amount of output going from the gateway to the browser.

JDBC Basics

We have now seen the configuration page and the output pages. It's time to look at the machinery in the middle that gets the data. In outline, JDBC code usually goes through the following steps:

- Load the database driver
- Open a connection to the database
- Create a `Statement` object
- Use the `Statement` object to send SQL statements to the database
- Process the results

I will now go through the steps in more detail, with simple code to illustrate them.

Load the Driver

The JDBC API is based on the notion of database-specific drivers that are manipulated by a `DriverManager` object.

Drivers come in several shapes and forms. Some drivers are freeware or open source; others are commercial products. The JDK comes with a generic **JDBC-ODBC bridge** that passes SQL statements on to an appropriate ODBC driver. This way, a Java application can work with any database for which an ODBC driver is available. JDBC drivers that talk to the database directly will generally provide better performance, though, so the bridge shouldn't be used in production environments.

A JDBC driver is loaded by calling the `forName()` method of the class called `Class`:

```
   String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
   ...
   Class.forName(driverName);
```

Objects of the Class class contain information about Java classes, such as the names of their methods and the arguments of their constructors.

> *For each Java class that has been loaded into the Java Virtual Machine there is a Class object. You can retrieve that object by calling the getClass() method on any instance of your class, or you can simply say myObj.class. Once you have obtained the Class object for your class, you have access to a lot of information about it.*

The forName() method dynamically loads a class that has not yet been loaded. Since a JDBC driver is a Java class, it can be loaded using the forName() method. All you need to know is the fully qualified name of the driver, which in our case is sun.jdbc.odbc.JdbcOdbcDriver.

Open a Connection

```
String dbURL = "jdbc:odbc:PhoneBook";
...
Connection con = DriverManager.getConnection(dbURL, "usr", "pwd");
```

Once the driver has been loaded, the getConnection() method of the DriverManager class will get you a connection to the database. The method takes three arguments: the database 'URL' in a driver-specific format, the username, and the password. The string that identifies the database usually starts with jdbc and uses colons to separate its various components. In the case of the JDBC-ODBC bridge, the second ('sub-protocol') component is odbc, and the third component is the ODBC data source name or DSN. A fourth component containing ODBC options may also be present.

Create and Use a Statement

```
String query String = "SELECT THENUMBER FROM PHONEBOOK WHERE THENAME='" +
   key + "';";

// If key is "Jane Doe" then queryString is
//  "SELECT THENUMBER FROM PHONEBOOK WHERE THENAME='Jane Doe';"
...
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(queryString);
```

Ultimately, we want to pass SQL strings to the database and have them executed. This functionality is wrapped in the Statement interface and its specialized variants, PreparedStatement and CallableStatement. This example shows the simplest approach. In many situations, prepared statements are a better choice because they are more efficient than plain Statements, and avoid the pitfalls in handling apostrophes in query strings (for example, if the name is 'O'Grady').

To execute a SELECT statement that returns data from the database, we run the executeQuery() method and place the results into a ResultSet object, as in our code. To execute an INSERT, UPDATE, or DELETE statement that changes the contents of the database, we run the executeUpdate() method that returns an integer: the number of affected rows. We also use executeUpdate() for executing data definition statements, such as CREATE TABLE. When used this way, the method returns a zero.

Process the Results

```
   if(null == rs) {
      return "db failure on " + key;
   }

   if(!rs.next())
      return "No such name as " + key;  // empty result set

   return rs.getString(1);
```

This code checks to see whether the `ResultSet` is `null` or empty. If it is neither, the code returns the first item in the `ResultSet` as a `String`. Typically, the result set contains multiple rows, and you can iterate through them in a loop:

```
   while(rs.next()) {
      /* Process the next row */
   }
```

Most of the `ResultSet` interface consists of `getXXX()` methods, where `XXX` stands for a data type, such as `String` or `Boolean`. All the 'get' methods take one argument that can be an integer (indicating the column number) or a string (indicating the column name).

The DBHandler Class

Our tool for database processing is the `DBHandler` class. There is a `DBHandler` object for each database that is used by the application within a user session. For each query available to the user, the `DBHandler` has a `Query` object, which is in effect a SQL query string with a name given to it; placeholders, to be replaced by the actual values coming from the user, fill some parameters of the query. You have seen such strings in `configure.jsp`:

```
   // queries available to the client
   String[][] dbQueries = new String[][]{

     {"AllTable", // all fields formatted as a table
      "SELECT * FROM FORECAST WHERE zip=?"},

     {"AllText",  // all fields formatted as a paragraph of text
      "SELECT * FROM FORECAST WHERE zip=?"},

     {"TimeTemp", // Date and temperature only
      "SELECT RecTime,Temp,DayLo,DayHi FROM FORECAST WHERE Zip=?"}
   };
```

`DBHandler` contains a `Hashtable` of such `Query` objects, created from the configuration file data. A `DBHandler` is created once per session; once it *is* created, it connects to the database (through a connection pool) and creates its `PreparedStatement` objects. After that, the program is ready to accept queries from the query entry page.

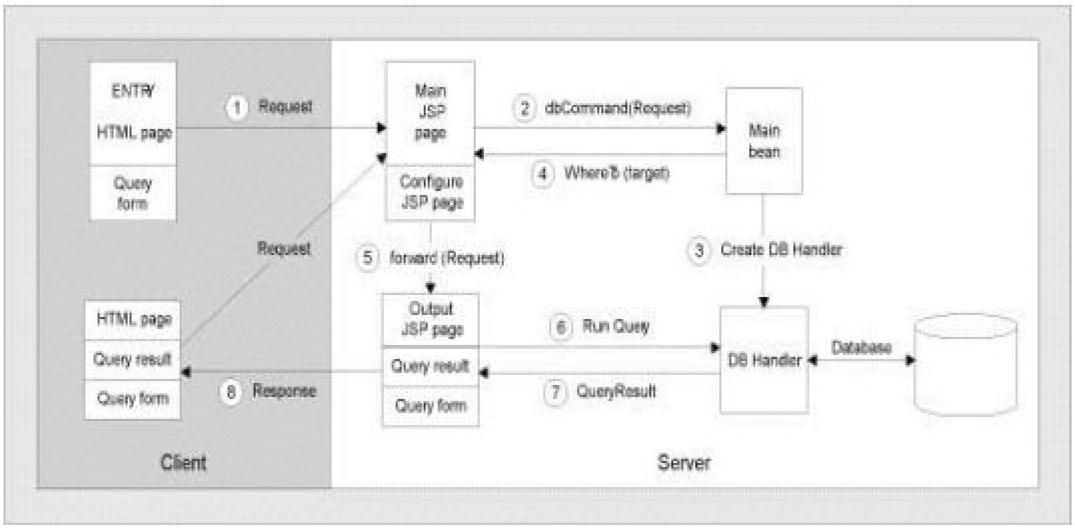
Let us walk through an example of a query. The `askCard` has the following code:

```
<card id="askCard" title="Weather">
   <do type="accept" label="Go!" >
      <go href="weather.jsp">
         <postfield name="query" value="$query" />
         <postfield name="QP1" value="$QP1" />
         <postfield name="target" value="wml-$(query)" />
      </go>
   </do>
   <p align="center">Weather Page</p>
   <p>
      Zip?
      <input name="QP1" format="*N" maxlength="10" value="" /><br />
      Query?
      <select name="query">
         <option value="TimeTemp" > time &amp; temp </option>
         <option value="AllTable" > all fields </option>
         <option value="AllText" > all fields-format </option>
      </select>
   </p>
</card>
```

To run the `AllText` query, for instance, the user fills in the `QP1` entry field with the zip value, and selects the query from the list of options. This information gets to the `DBHandler` via the `HttpServletRequest` object and the main bean. The `DBHandler` retrieves the prepared statement using the name given, replaces its question marks with the value of query parameter, and runs the query.

How the Query is Run

The main bean has a method for executing the query, `queryResult()`, but it is not called from the bean itself: it is called from the target page to which the request is forwarded. Here's the diagram again:



The `queryResult()` method, in turn, calls an output method of the `DBHandler`. The `DBHandler` has several such methods that differ in how the result set is packaged: it can be a two-dimensional array of `String`, or a lazily evaluated sequence. In our application, `QueryResult` is sent as a string matrix to the output pages that you have already seen.

Our interest here is in flexibility: How is this customizable? A minor customization would be to give the user more queries within the same application; a major customization would be creating a different application.

Updating the Database and Adding Queries

In order to be able to update the `FORECAST` table, you don't need any new Java code: you simply use an update SQL query instead of a `SELECT` one. Here are the additions that need to be made:

1. Add a query to the query list in `configure.jsp`:

```
      {"addOne",
       "INSERT INTO FORECAST VALUES(?, ?, Now, ?, ?, ?, ?, ?, ?, ?)"},
```

Here, the `?` symbols are placeholders for field values, and `Now` is a (MS Access-specific) way of referring to the current date.

2. Add a new response target to the target list in the same file:

```
      {"addOne", "/jsp/weather/xhtml/TimeTemp.jsp"},
```

We can use the same `TimeTemp.jsp` because (like `AllTable.jsp`) it just loops through whatever is in the string matrix. In this case, it will be a single column, with the second row showing an integer — the number of rows affected.

3. Add eight more input elements (postfields) to the form. The new elements are for query parameters and therefore named QP2, QP3, and so on. (There are ten fields altogether, and one of them is the current date.)

4. Finally, we add another option to the select element:

```
         <option value="addOne">addOne </option>
```

If you enter data in the text boxes and select addOne from the list of available queries, this is the response you will see:

```
Weather! Stick your hand out to see if it's raining!<br />
<table border="1">
   <tr>
      <td>NumberOfRowsAffected</td>
   </tr>
   <tr>
      <td>1</td>
   </tr>
</table>
```

The weather report for the zip code that you entered has been updated, as you can verify by running a TimeTemp query. Of course, in real life it is unlikely that the updates will be entered manually from a form (although one can think of a junior high school project to keep the school's weather page up to date). More likely, they will come from another application. This is easy to integrate into our framework.

Adding Queries

The update query we have just discussed is an example of what you need to do to add queries in our framework. Here is a summary of how add a new query:

- Write the query in SQL, with question marks for parameters to be entered by the user
- Give the query a name
- Modify the entry form in two ways:
  - Add the new query to the SELECT element
  - Add entry elements if the new query has more parameters than previously provided for
- Modify `configure.jsp`, in two ways:
  - Add a new item to the query list (a query name-query SQL string pair)
  - Add a new item to the list of output JSPs
- Create an output JSP for the new query

A new query is then ready for use. As we discussed earlier, in a 'real' framework you would edit an XML configuration file instead of modifying `configure.jsp`, but editing `configure.jsp` works fine.

Changing the Application

Now, instead of modifying an application, consider what's involved in replacing it with another one that is similarly structured. Suppose that instead of a weather report, you want to build a "ride board": a web site where people go either to post or to look for rides (or lifts, depending on which side of the Atlantic you're on) from one place to another. Our framework is generic enough that you would not have to change any Java code. All you would have to do is:

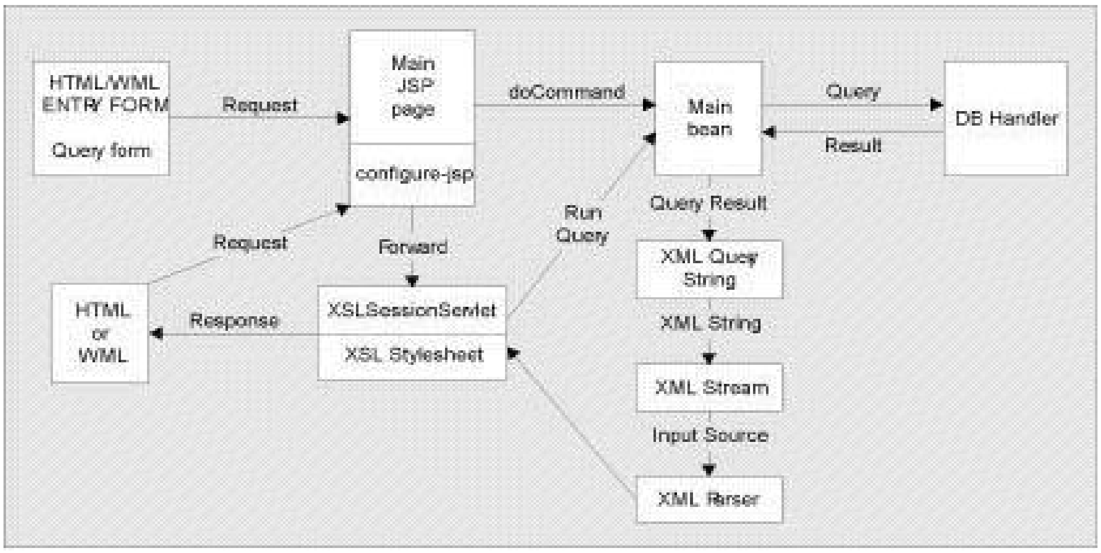
- Provide a different database
- Replace the configuration file
- Provide appropriate WML decks

We go through this exercise in our chapter in *Professional WAP* (Wrox Press).

Web-WAP with XSLT

The following diagram describes a revised XSLT-based system for Web-WAP generation.

> *The system uses James Clark's xt, with minor modifications to two of its classes. The modifications are briefly explained below, and in more detail in our chapter in* Professional JSP Programming *(Wrox Press). The code for the chapter (and this presentation) is downloadable from the Wrox web site.*



The main addition to the previous design is the "conversion module" in the center that converts the result of the query into something that an XML parser can parse and deliver to the XSLT processor. Apart from that, the changes that need to be made are as follows:

- The configuration page, `configure.jsp`, will show more targets.
- Within the stylesheet, a major addition is the mechanism for running Java methods from within XSLT. (It is the stylesheet, not a JSP, that calls `qBean.queryResult()` in this version of the framework.)
- The main bean has minor additions to make the session and session ID available to the stylesheet, and to replace the mechanism for controlling the size of output. The `getDef()` method is no longer helpful because the stylesheet will be getting its information from parsed XML nodes rather than from a `Dict`.
- There are two minor revisions of classes provided within xt.

The Configuration Page

The configuration page, you will recall, is for a systems administrator to edit in order to configure and maintain the system. One way to modify it would be to double the number of targets available to the user at any given time. This would make the user responsible for the choice between the system that uses JSP, and the system that uses XSLT. It seems a better division of responsibilities to give this task to the system administrator by providing a single Boolean switch between the two systems. The assumption is that at any given time, only one of the two systems will be in use, and that switching between them will be infrequent.

All the changes are in the "targets" section; the rest of the page is the same and not repeated here:

```
boolean sendToXSLNotJSP = false; // the switch, initially set to do JSP

String[][] responseTargets;
if(!sendToXSLNotJSP)
   responseTargets = new String[][]{ // default targets for a JSP system
      {"error", "/jsp/weather/errorpage.jsp"},
      {"AllTable", "/jsp/weather/xhtml/AllTable.jsp"},
      {"AllText", "/jsp/weather/xhtml/AllText.jsp"},
      {"TimeTemp", "/jsp/weather/xhtml/TimeTemp.jsp"},
      {"wml-AllTable", "/jsp/weather/wml/AllTable.jsp"},
      {"wml-AllText", "/jsp/weather/wml/AllText.jsp"},
      {"wml-TimeTemp", "/jsp/weather/wml/TimeTemp.jsp"}
   };
else
   responseTargets = new String[][]{ // targets for an XSLT system
      {"error", "/jsp/weather/errorpage.jsp"},
      {"AllTable", "/xslAllTable/xslrules/weather.xml"},
      {"TimeTemp", "/xslTimeTemp/xslrules/weather.xml"},
      {"AllText", "/xslAllText/xslrules/weather.xml"},
      {"wml-AllTable", "/xslWmlAllTable/xslrules/weather.xml"},
      {"wml-AllText", "/xslWmlAllText/xslrules/weather.xml"},
      {"wml-TimeTemp", "/xslWmlTimeTemp/xslrules/weather.xml"}
   };
```

The new targets indicate that there are six stylesheets in the `xslrules` directory: three for XHTML output, and three for WML output. Also in that directory is an XML file, `weather.xml`. In a framework that uses XML more (for configuration, or data interchange, or both), this file could contain useful information. In this example, it is simply ignored, as the stylesheet synthesizes its output from the result of the query.

xslTimeTemp.xsl

The important part of this stylesheet (and the other two stylesheets) is how Java code gets called. First, in the declaration part, a special namespace is declared:

```
xmlns:xqd="http://www.jclark.com/xt/java/MyNa.jspUtil.XmlQueryStringDoc"
```

The xqd namespace (a mnemonic for XmlQueryDoc) is declared to be associated with the `XmlQueryStringDoc` class in the `MyNa.jspUtil` package. This is the class that runs the query, converts its `QueryResult` to an XML string and passes it on to an XML parser. This is all done by a static `queryResult()` method that returns a `NodeIterator` object, which is the kind of object that xt expects from its parser.

We are going to use the `queryResult()` method within the stylesheet. Java methods are a particular kind of what are called 'extension functions' in the XSLT recommendation, which says that although the 1.0 version of the XSLT does not define a standard mechanism for implementing such extensions, "the XSL WG (working group) intends to define such a mechanism in a future version of this specification or in a separate specification." There have been more recent discussions that indicate that the ways Java extensions are implemented are very similar in all the major XSLT processors, so we can expect a standardization effort soon.

In outline, the stylesheet is structured as follows:

```
<!-- stylesheet declarations -->

   <xsl:template match="/">
      <!-- matches the root of the XML document tree -->

      <!-- first block of HTML or WML material to pass through to output -->

      <xsl:variable ... />
      <!-- declares and initializes a variable -->

      <xsl:choose>
         <!--
            checks to see if there is output;
            if not, outputs a message; otherwise outputs a table
         -->
     </xsl:choose>

      <!-- second block of HTML or WML material for output -->

   </xsl:template>
</xsl:stylesheet>
```

Here is the stylesheet for WML output, broken into sections. The first one contains declarations:

```
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xqd="http://www.jclark.com/xt/java/MyNa.jspUtil.XmlQueryStringDoc"
          xmlns="http://www.w3.org/TR/xhtml1/strict"
          xmlns:javaout="http://www.jclark.com/xt/java"
          exclude-result-prefixes="javaout xqd #default">

   <xsl:output method="javaout:MyNa.jspUtil.XMLOutputHandler"
      indent="yes"
      encoding="UTF-8"
      media-type="text/vnd.wap.wml"
      omit-xml-declaration="no"
      doctype-public="-//PHONE.COM//DTD WML 1.1//EN"
      doctype-system="http://www.phone.com/dtd/wml11.dtd" />

   <xsl:param name="theSessionID" select="NoHttpSessionIDProvided"/>
```

Next we have the initial template, including the Java function call:

```
   <xsl:template match="/">
      <wml>
         <head>
            <meta http-equiv="Cache-Control" content="no-cache" forua="true"/>
         </head>
         <card id="output" title="AllTable">
            <do type="accept" label="again" >
               <go href="#askCard" />
            </do>
            <xsl:variable name="rows"
               select="xqd:query-result($theSessionID)//*[name(.)='row']" />
```

The `xsl:`choose element is next. For small WAP screens, we format the output as two columns:

```
            <xsl:choose>
               <xsl:when test="count($rows)=0">
                  <p mode="nowrap">
                     Sorry, no weather in your zip-code.
                  </p>
               </xsl:when>
               <xsl:otherwise>
                  <p>
                     <table columns="2">
                        <!-- a row for each header -->
                        <xsl:for-each select="$rows[1]/*" >
                           <tr>
                              <td> <xsl:value-of select="@*" /> </td>
                              <td> <xsl:value-of select="." /> </td>
                           </tr>
                        </xsl:for-each>
                     </table>
                  </p>
               </xsl:otherwise>
            </xsl:choose>
         </card>
```

The rest is mostly the entry card in pure WML:

```
         <card id="askCard" title="Weather">
            <do type="accept" label="Go!" >
               <go href="weather.jsp">
                  <postfield name="query" value="$query" />
                  <postfield name="QP1" value="$QP1" />
                  <postfield name="target" value="wml-$(query)" />
               </go>
            </do>
            <p align="center">Weather Page</p>
            <p>
               Zip?
               <input name="QP1" format="*N" maxlength="10" value="" /><br />
               Query?
               <select name="query">
                  <option value="TimeTemp" > time &amp; temp </option>
                  <option value="AllTable" > all fields </option>
                  <option value="AllText" > all fields-format </option>
               </select>
            </p>
         </card>
      </wml>
   </xsl:template>
</xsl:stylesheet>
```

Resources and Future Developments

This final section lists resources of two kinds: specific resources for the software shown in the talk, and information to enable you to follow future developments.

First, here are the URLs for software downloads:

- To download JAXP, go to java.sun.com/xml
- To download xt, go to www.jclark.com/xml
- To download Tomcat, go to jakarta.apache.org
- To download all the code for this example, go to www.wrox.com, looking for the code for Professional JSP Programming

Both XSLT and JSP are rapidly evolving technologies. Within XSLT, you can expect a standardization of the extension-function mechanism for calling Java methods from within an XSLT stylesheet. At present, the same code is portable between xt and Michael Kay's Saxon. Watch w3c.org/tr for new developments.

On the JSP side, three important sites to watch are:

- java.sun.com/jsp, for the latest JSP developments in general
- jakarta.apache.org, especially for Tomcat and Struts projects
- xml.apache.org, especially for the Cocoon project

As far as specific technologies are concerned, we want to pick out two:

- The `WebRowSet` implementation currently in early development on Java Developer Connection (http://developer.java.sun.com/developer/) outputs its populated rowsets as XML if needed, which gives you valid XML for your XSLT processor direct from the database. It also allows you to get a `Query` result in one call, and then release the connection but allow the `RowSet` to remain through the session.
- The JSP custom tag libraries from JRun and Apache have an XSLT tag that will apply the stylesheet specified in the URL to the XML in the JSP. That's middle ground between the two architectures presented in this talk.

Summary

In this article, we have introduced:

- Architectures for JSP applications
- Basic JSP and JDBC
- Generating WML (and other XML) content from database queries using JSP
- Customizing the application and creating a new one
- WML and XHTML content from the same data source using XSLT-controlling Java for output

The main point of this article has been to bring up several quite general design ideas, and to create a framework for generating both XHTML and WML content from the same data source.

In the actual application we looked at, the data source is a relational database, but it doesn't have to be: the framework is general enough that anything wrapped in a `Properties` object or described by an XML document can be a source of data.

We did not have time to explore the data connection in depth because our task has been to explore and compare the output methods: JSP and XSLT. Can we say that one is definitely better than the other?

In truth, there is probably not yet enough accumulated experience to pass definitive judgment, especially since in both cases we are talking about rapidly evolving technologies. Both JSP and XSLT are extremely powerful tools that are certainly up to the job. For applications that are maintained by programmers, JSPs are probably a better choice, if only because they are needed anyway — why bring in yet another set of tools? However, XSLT enables non-programmers to do very impressive things, so in an organizational context where extensibility by non-programmers is important, XSLT may prove preferable.

In the meantime, the framework presented here provides a test bed for more exploring and experimentation. Enjoy!

Acknowledgements

I would like to acknowledge the contributions of my business partner and co-author Tom Myers. Everything here is our joint effort.