WebSphere Development

Enhancing J2EE Applications with Ajax

An example using the IBM® WebSphere® Application Server Feature Pack for Web 2.0

Kevin Haverlock
WebSphere Application Server Development, Ajax Technologies
kbh@us.ibm.com

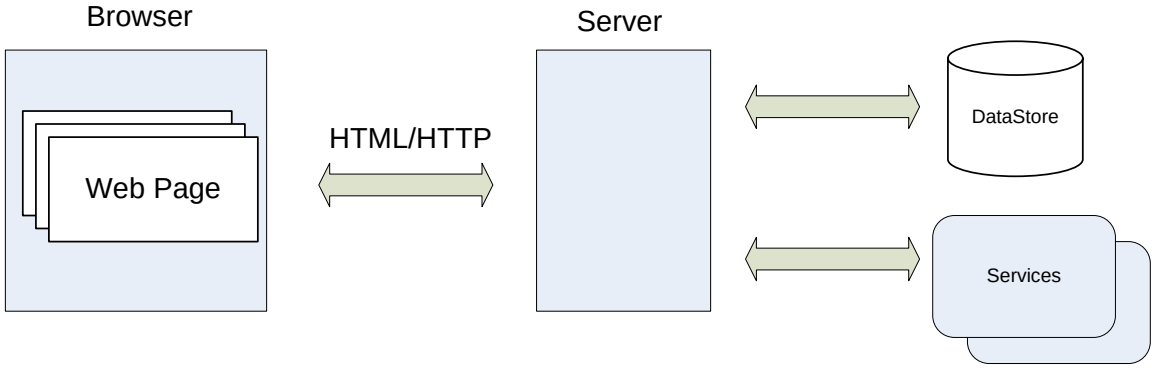
Ajax represents an evolution of existing Web development that lends itself to the creation of rich user interface presentations on the browser. The dividing line is becoming blurred between the capabilities of a heavy weight graphical user interface application running on the native operating system and the richness possible within today's browsers. One does not have to look much further than Google Maps and the newly renovated Yahoo! Mail to see how Ajax technology is bearing out.

The continued innovation in browser based user interfaces have caused customers to want these kinds of capabilities with the Web sites they interact with. Ajax development is an exciting opportunity to create new and innovative approaches to delivering content to users.

At the same time, how does one achieve this in the context of today's J2EE applications? This article will look at how an existing J2EE application was enhanced using IBM's recent feature pack for Ajax development on WebSphere Application Server.

Figure 1.0 shows a typical Web-based application pattern that is very common in a Web development scenario. The server accepts requests from the browsers and issues a response back to the client. In the case of a browser client, the browser sends the request and waits for the response. The response is normally HTML which is nothing more than meta-data containing the presentation information of how the page should be laid out in the browser. There may be additional page styling that is returned in the form of Content Style Sheets (CSS). The creation of the content is done on the server and returned to the browser. The content of the page is often aggregated from a number of back-end sources. These may be additional Web services requests to other domains or content from persistent data stores such as DB2, Derby, or MySql. While there may also be JavaScript present in the page returned to the server, it normally is used in a supporting role to make the page seem more interactive.

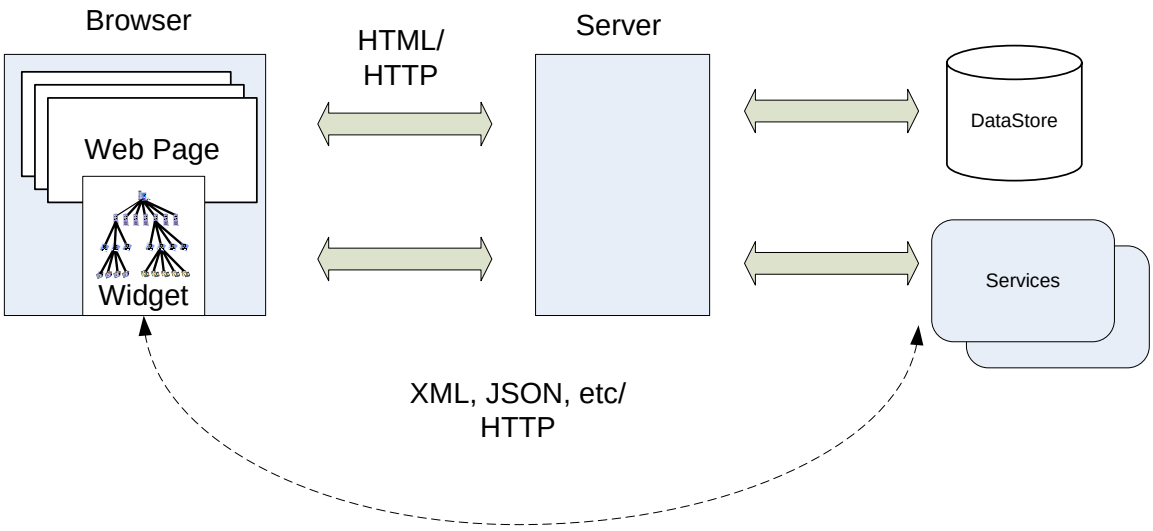
Figure 1.0: A-typical Web architecture



An Ajax design pattern allows the shifting of the balance of the user interface to the browser client. The creation of the viewable content can be shared between the browser

and server. Some Web applications have the majority of the presentation done within the browser. An example would be Google Docs Beta which provides spreadsheet, documentation, and presentation functionality in a form one would expect to find running on a local operating system. These applications mix customized widget code that executes on the client with static HTML. Figure 2.0 shows how Ajax can impact the typical Web based application.

Figure 2.0: An Ajax enhanced Web architecture



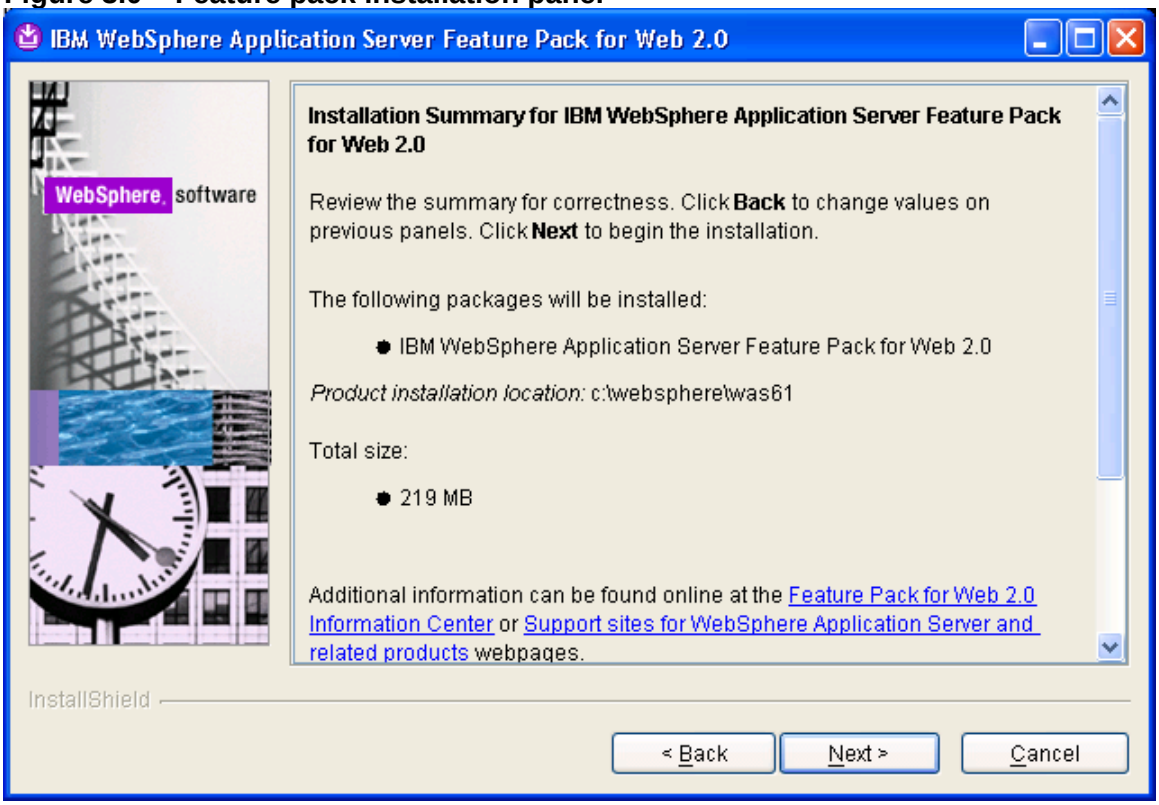
In an Ajax model, creation of content in the browser is done by the browser locally. In JavaScript this done by manipulating the Document Object Model that the browser maintains to describe the document the user is viewing. Updates to the DOM are immediately reflected in the presentation that the user sees within the browser. On the browser side the constructs are known as 'widgets' and are used to described self contained code that can be used to manipulate the presentation, react to user input, or asynchronously communicate back to the server.

With aspects of the presentation delegated to the browser, the browser now needs a way to communicate back to the server to derive information. As an example, the widget may have a Web table to update with the data representing rows in a database server. In JavaScript, a powerful API can be used called XMLHttpRequest (XHR). An XHR request allows the establishment of an independent communication channel between the server and browser page the user is viewing. The API allows the transfer of XML or other text information using HTTP. The server treats the request from the client as it would a normal request and returns a response. The response can contain the data the widget needs to display in the browser.

A look at the IBM WebSphere Application Server Feature Pack for Web 2.0

The IBM WebSphere Application Server Feature Pack for Web 2.0 provides technology that can be used to create Ajax styled architectures. The feature pack is available with WebSphere application server 6.1, 6.0.2 and WebSphere Community Edition 2.0. The feature pack's functionality is intended to provide developers and architects resources to create Ajax styled Web applications and architectures. The feature pack includes both client-side runtime and server-side functionality. Figure 3.0 shows the installation panel for the feature pack.

Figure 3.0 – Feature pack installation panel



The Client-Runtime – Dojo Toolkit and Extensions by IBM

The Client-Runtime included with the feature pack consists of the technologies that are running on the browser-client. They include the open source Dojo Toolkit and a set of IBM extensions to the Dojo Toolkit to support additional functionality.

The Dojo Toolkit 1.0 (www.dojotoolkit.org) is a powerful open-source JavaScript library that can be used to create rich and varied user interfaces running within a browser. The library requires no browser-side runtime plug-in and runs natively on all major browsers. This is boon for JavaScript developers since it helps abstract away the eccentricity of different browser implementers.

The open-source Dojo Toolkit provided with the IBM's feature pack is divided into five sections:

Base & Core
The Base is the kernel of the Dojo Toolkit and consists of dojo.js. The file is designed to be compact and optimized so as not take long to download to the browser. It contains the bootstrapping, useful utilities, event notification, to name just a few items.

The Core, contains wide variety of graphical user interface widgets and the IO Transport for XHR requests to the server.

Dijit
Dijit builds on the Base and Core by providing a rich set of additional widget controls. The controls are internationalized and accessibility enabled.

Dojox
Dojox contains experimental aspects of the Dojo Toolkit and represents innovative material that may some day move into the base or Dijit modules. Dojox is an incubator of sorts and a preview of new features. Some of the modules located in Dojox include charting, offline storage, and grid to name a few.

Util
The Util contains a testing harness for Dojo and can be used to test the widgets that are provided with the Dojo Toolkit.

The best way to begin to get a feel for the Dojo Toolkit is to experiment with the test samples that are provided. The samples can be opened directly within the browser and offer a glimpse at the incredible flexibility and creativity for creating your own custom widgets or using what is already provided in the toolkit.

IBM Extensions to the Dojo Toolkit

In addition to the open-source Dojo Toolkit for creating rich client side applications, IBM also provides a set of JavaScript extensions that developers will find useful.

Atom Feed Widget

The Atom library is a client-side widget that can be used to render and use Atom syndication feeds. The library contains sample components to help developers utilize Atom feeds with their code:

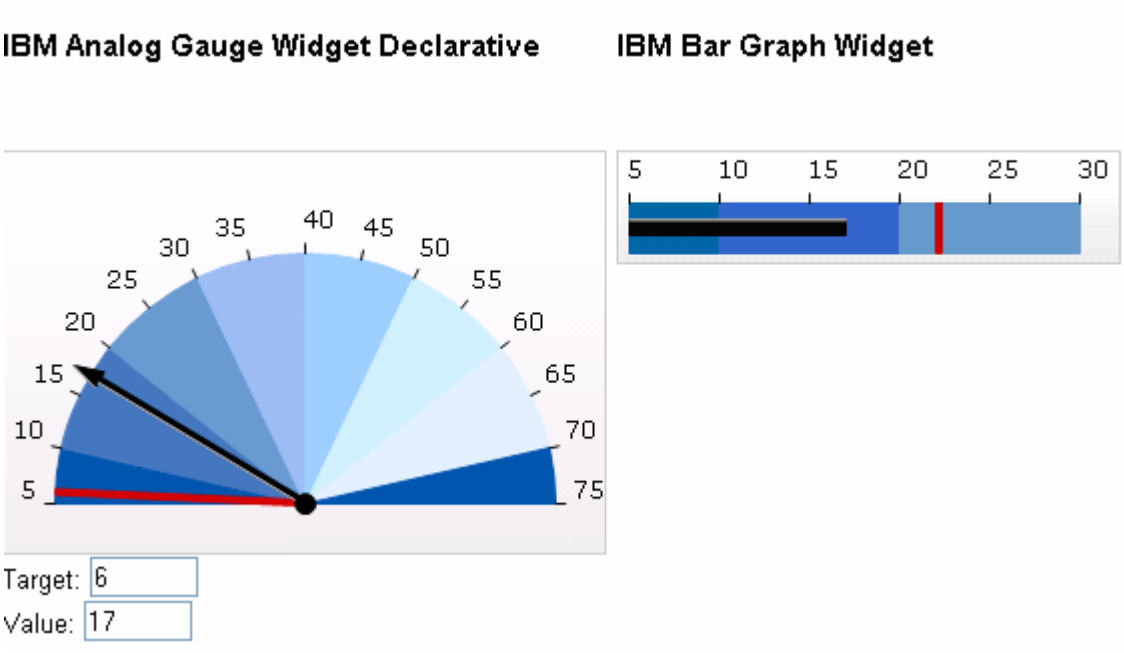
- A base library, supplying utility functions, an implementation of the Atom data model as JavaScript objects, and a wrapper object to handle two-way communication with an Atom feed
- The AppStore, an implementation of the dojo.data APIs that provides a data storage solution supported by an application server

IBM Gauge Widgets

The gauge library includes a pair of widgets for displaying numerical data in a graphically rich way. Using Scalable Vector Graphics (SVG) or Vector Markup Language (VML), depending on the browser, the AnalogGauge and BarGraph widgets display numerical data with customizable ranges, tick marks, and indicators at any size. The Gauge Widget can be used to create dynamically self-updating graphical displays and dashboards.

As with the Atom library, the gauge library comes with several examples and test cases to show its capabilities. Figure 4.0 shows an example of the IBM Gauge library.

Figure 4.0 – IBM Gauge and Bar Graph widgets



IBM SOAP

The IBM SOAP extension can be used to connect a client-side browser widget to an existing SOAP-based service. The overhead of creating a SOAP envelope is handled by the extension. Remote procedure calls to invoke the SOAP service are also handled by the extension.

IBM Open Search library

The IBM Open Search extension makes it easy to invoke any Open Search-compliant service and to bind search results to widgets within your Ajax application.

Server-side libraries and connectivity

While the discussion of the IBM feature pack above describes the client-side runtime, there is rich set of libraries and connectivity features provided on the server to assist in client development. The features include:

The Ajax proxy

The feature pack provides a Servlet based forward proxy that can be used in the aggregation of content from different sites. To provide control, the proxy contains a white-listing configuration file that can be used to define the sites the proxy can access. Additionally, the proxy can filter on HTTP headers, cookies and mime-types to provide a level of control over the sites that a browser-based client can access.

Web-remoting for Java Components

A challenge in combining Ajax style architectures and J2EE is mapping client-side runtime to J2EE constructs. As an example, consider a JavaScript widget that displays information in a table that is dynamically created using JavaScript. The data needed for the table exists on the server and is accessible using EJBs, how does one access those EJB constructs?

The feature pack provides a Remote Procedure Call Adapter (RPCAdapter) that is provided as a JAR library which can be embedded into a server-side Web application. The RPCAdapter can be used to accept HTTP requests such as POST and GET and map the requests directly to user created classes. One of the powerful aspects of RPCAdapter is the ability to serialize EJB session and Collection data to a JSON or XML stream returned to the browser client. The JSON and XML data can contain the information to be displayed by the widget.

Apache Abdera libraries
Apache Abdera is an open-source project providing feed syndication support. Abdera addresses both the Atom syndication format and the Atom publishing protocol. The Abdera libraries can used on the server to read syndication feeds from other sources or to generate your own feed content for use by your widgets.

JSON4J
The JSON4J library is an implementation of a set of JSON handling classes for use within Java environments. The library can be used to derive your own JSON data streams. The JSON4J library provides the following functions:

- A simple Java model for constructing and manipulating data to be rendered as JSON

- A fast transform for XML-to-JSON conversion. JSON4J can be used to convert an XML reply from a Web service into a JSON structure for easy use in an Ajax application.

  The advantage of the transformation is that Ajax-patterned applications can handle JSON formatted data without having to rely on ActiveX objects in Microsoft® Internet Explorer XML transformations and other platform-specific XML parsers. In addition, JSON formatted data tends to be more compact and efficient to transfer.

- A JSON string and stream parser that can generate the corresponding JSONObject, which represents that JSON structure in Java.

Web messaging service
The Web messaging service uses a publish and subscribe pattern to connect the browser to the WebSphere Application Server Service Integration Bus for server-side event push to the browser. Client-server communication is achieved through the Bayeux protocol. You can consider the Web messaging service implementation as a comet server implementation. The Dojo Toolkit provides client-side support.

Currently, the Dojo Toolkit is the only JavaScript library to support the Bayeux protocol, although any JavaScript library that implements Bayeux protocol support can communicate with Web messaging service. The Web messaging service server bridges browser clients to the Service Integration Bus, allowing a Web service or any other item connected to the bus to publish events to Web-based clients. You can use the Web messaging service in a new or existing application by placing a utility file library JAR in an application Web module, setting up a simple configuration file, and configuring Servlet mappings. The Web messaging service is included in the Quote Streamer for WebSphere Application Server product samples.

Putting it together: A J2EE Example

At this point, the article has discussed a typical Ajax based architecture and the motivation and features behind the feature pack for IBM WebSphere. Let's look at how you might put it together into an existing J2EE application.

Plants By WebSphere is among a number of samples that are provided with the IBM WebSphere Application Server Feature Pack for Web 2.0. The sample application represents a fictitious online Plant store where one can order and purchase flowers, trees, vegetables, and accessories. Figure 5.0 shows the front page of the application.

Figure 5.0 – Front page of PlantsByWebSpereAjax

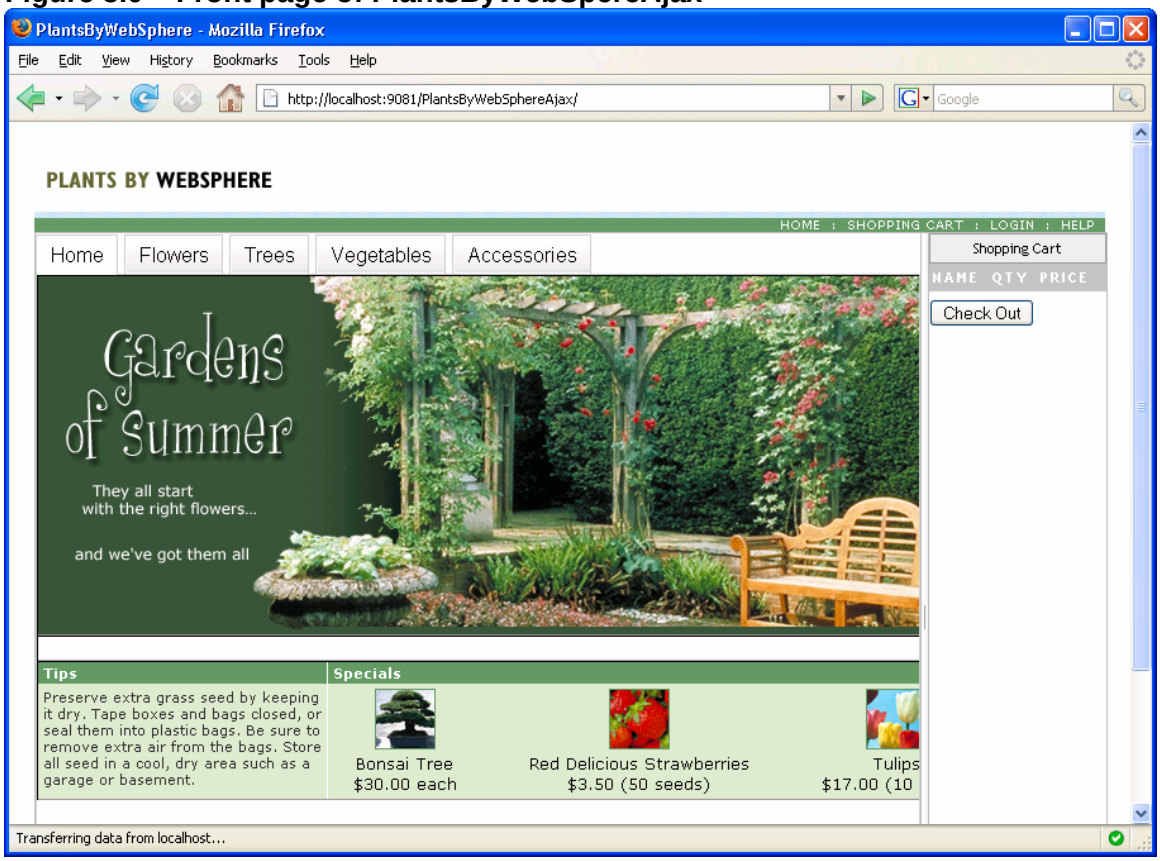


Figure 6.0 describes the architecture of the application in it's original form before attempting to add Ajax styled features. The architecture is intended to be fairly typical for a J2EE application running on WebSphere Application Server. At a high level, the application adheres to a Model-View-Controller design pattern which most Web applications follow on some level. A Browser accesses the URL for the application which returns a JSP rendered HTML page. Additional requests are issued to the Web application from the browser and Servlets are used to control the flow as users move through the purchase request. EJBs are used to serve model data available on the database.

Figure 6.0: A-typical Web architecture for Plants By WebSphere

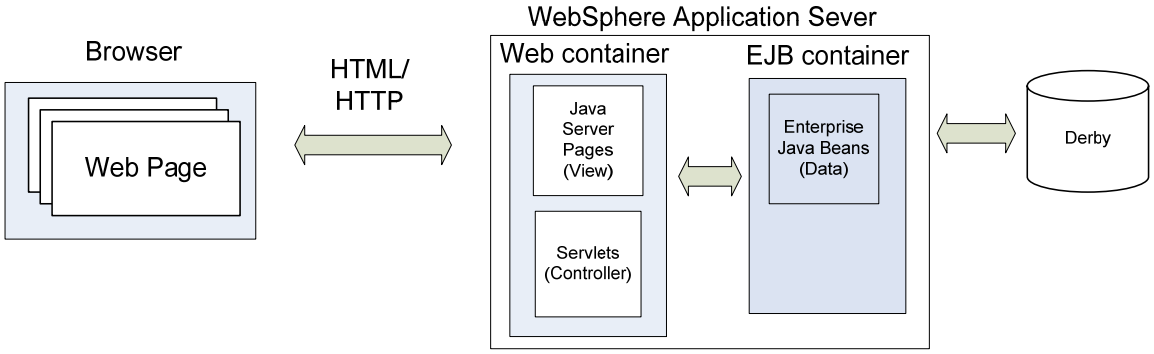
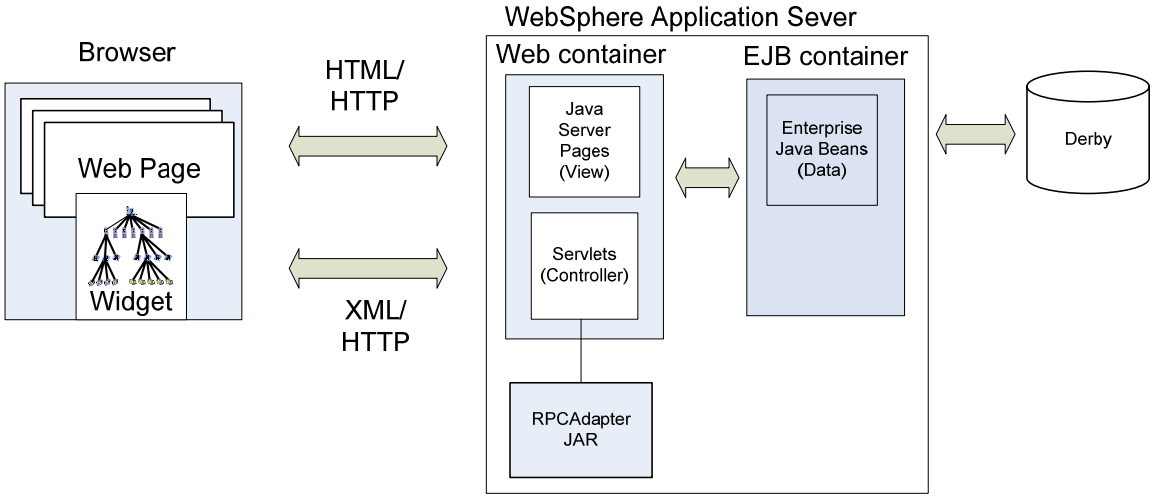


Figure 7.0 shows how the architecture of the application has been augmented using Ajax.

The intention is not to rewrite the application, but rather take advantage of technologies in the IBM feature pack that would improve and create a more interactive and rich experience for the user.

Figure 7.0: Enhancement with Ajax and feature pack functionality



On the browser side, the application used widgets provided with the Dojo Toolkit. Additionally, customized user interface widgets were created to improve the interactive nature of Plants By WebSphere while not rewriting it. The customized user interface widgets are asynchronous, meaning they communicate using the browsers XHR mechanism supported by the Dojo Toolkit. The widgets use an XML interchange format to exchange data with the server. On the server, the RPCAdapter provided with the feature pack is used to convert EJB data into an XML interchange format that can easily be consumed by your newly created widgets on the browser.

A characteristic of Ajax applications is an improved response on the user interface. Plants By WebSphere uses the Dojo Toolkit within the Web browser to improve the User

interface of the application. The Dojo Toolkit is pure JavaScript and the JavaScript files can be hosted directly in the Web-Content directory of the Web Archive File (WAR) or exist as static Web content on a performance optimized content delivery network. As a sample, the Dojo Toolkit JavaScript files are hosted as part of the WAR.

The Dojo Toolkit supports being used declaratively or procedurally. In a declarative role, you inline the JavaScript you plan to use directly within the HTML content. The Dojo Toolkit contains a wealth of widgets that can be used declaratively within HTML reducing the need for you to hand code the function. In the case of Plants By WebSphere, Dojo Toolkit widgets are directly embedded into the JSP pages.

An Example: Web Form Handling

A common scenario in Web applications is Form handling. The user enters data into a Web-page such as name, address, preferences, and so on. The information is then sent back to the server for processing and the result returned to the user. A common action on the form is validation to ensure the content the user is entering is correct. Was a number entered in place of a letter? did they enter a correct zip code?. The Dojo Toolkit provides a rich set of form handling validation that can be added to Web pages. Listed below is an example that is used within the Plants By WebSphere application and represents a typical example of how to declaratively use the Dojo Toolkit within a Web application.

Let's begin by declaring the usage of Dojo within your HTML page. Looking at listing 1.0, the first <script> tag declares usage of dojo.js. Dojo.js is the core Dojo Toolkit kernel and is required when one use the Dojo Toolkit. The second <script> tag declares the dojo widgets that the page uses. The dojo.declare clause compares to the Java import or the C++ includes clause. The statement tells the dojo.parser where to locate the appropriate Dojo Toolkit JavaScript file the page uses. For this example, the page uses the dijit.form.ValidationTextBox widget. Additionally, the JavaScript dojo.parser needs to be included as well. The parser is used to scan the page for widgets to declare. Listing 2.0 will explain why.

Listing 1.0: Declaring the usage of Dojo's ValidationTextBox

```
<title></title>

<script type="text/javascript"
        src="/PlantsByWebSphereAjax//dojo/dojo.js"
        djConfig="isDebug: false, parseOnLoad: true,
        extraLocale: ['de-de', 'en-us']">
</script>

<script type="text/javascript">
        dojo.require("dijit.form.ValidationTextBox");
        dojo.require("dojo.parser");// scan page for widgets and
                                    // instantiate them
</script>
```

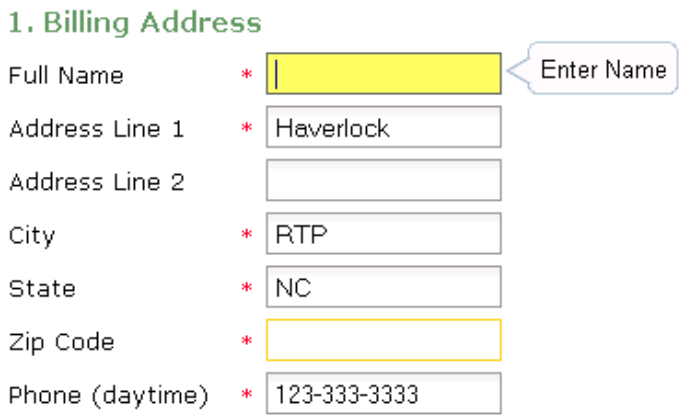
Listing 1.0 showed how the Dojo Toolkit is declared within your JavaScript and the type of widgets the page will use. Listing 2.0 shows how the widget is declared within the page. The dojo.parser that was declared in Listing 1.0 will be used to scan the page and inline the necessary JavaScript code from the Dojo Toolkit to enable dynamic form validation.

Listing 2.0: Declaring the use of Dojo's ValidationTextBox widget

```
<TD width="100%">
   <P>
      <input type="text" id="sname" name="_sname" class="medium"
             dojoType="dijit.form.ValidationTextBox"
             propercase="true"
             required="true"
             promptMessage="Enter Name" />
   </P>
</TD>
```

Figure 6.0 shows the result. When the user fails to enter a value for a required field, the browser will dynamically inform the user.

Figure 6.0



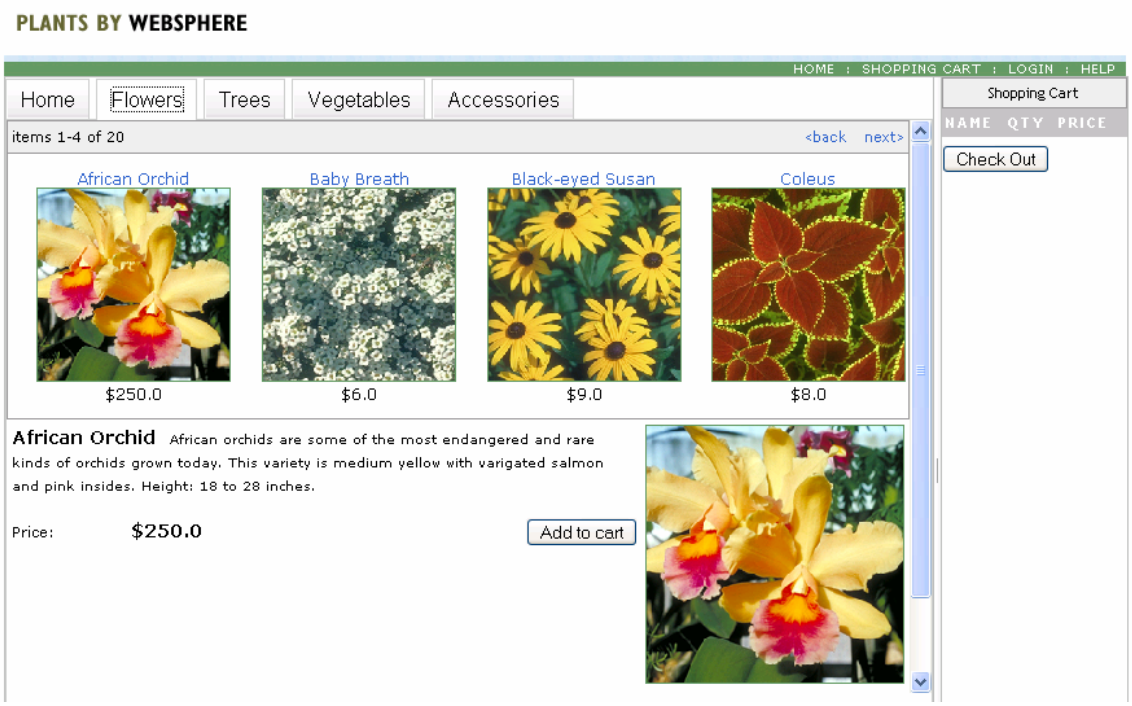
Form validation is a simple example of how one can be begin to use Ajax within an existing J2EE application. At the same time, it represents a powerful example of how seamlessly a user interface can be improved without having to rewrite a ton of existing code.

There are numerous other examples of how declarative forms of programming using the Dojo Toolkit can be used within a J2EE application. Dojo ships a wide array of sample applications within the test directories of the Dojo Toolkit. These applications will run within a browser by opening the HTML page.

Creating your own custom widgets
Another interesting example is creating your own customized JavaScript widgets. Dojo provides a powerful framework to create innovative widgets that can be embedded directly within your HTML pages.

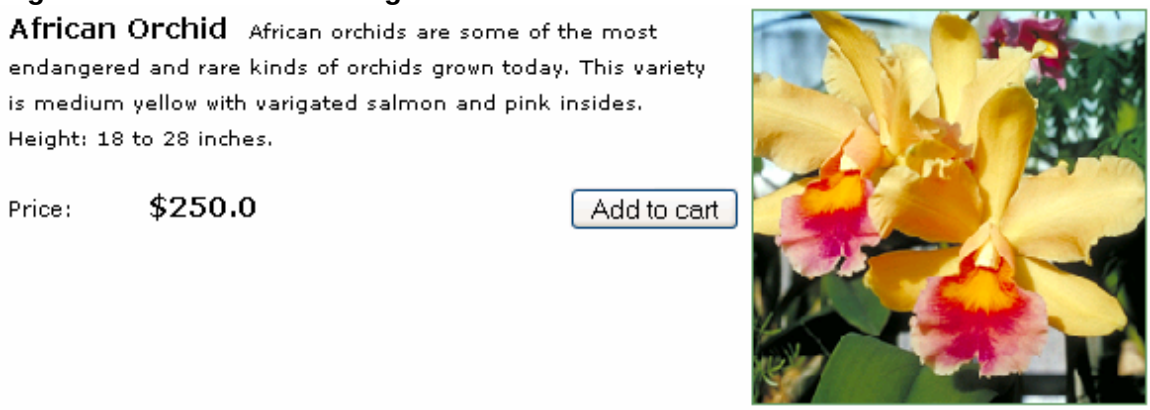
Figure 7.0 shows the catalog browsing page for the Plants By WebSphere application. The catalog page shows pictures of items across the top and detail information of the item at the bottom when the user clicks one of the images.

Figure 7.0: Catalog page for Plants By WebSphere



While this page consists of a number of different widgets, let's look at the ItemDetails Widget at the bottom in figure 8.0. The itemDetail widget fetches information from the server and displays it in the details window. The user can add the item to the cart or drag it to the cart using the left mouse button. The widget fetches information from the server when the user clicks on one of the catalog items at the top. Let's look at how the ItemDetail widget was created.

Figure 8.0: ItemDetails Widget



When creating widgets, there are three files one needs to consider. The first two are the HTML and CSS template files. These will be used to define the skeleton of what the page will look like when the widget is rendered within the browser. The HTML page will contain declared attachment points where the DOM elements will be inserted by your widget. The attachment points are called DojoAttachPoint.

Listing 3.0 shows a portion of the ItemDetail widget's HTML template and the DojoAttachPoint for the table rows. As the JavaScript in the customized widget process content, it will dynamically create elements and insert them into your attachment points for the DOM.

Listing 3.0: dojoAtachPoint's in ItemDetails.html Dojo Template

```
<div>
<table cellpadding="2" cellspacing="2" border="0" width="100%"
       height="100%">
   <tr width="100%" height="100%">
      <td valign="top" align="left" width="75%">
         <span class="itemNameText"
               dojoAttachPoint="nameElement"></span>

         <span dojoAttachPoint="descElement"
               class="itemDescText"></span><br><br>

         <table cellspacing="0" cellpadding="0" width="100%"
                height="100%">
            <tr style="width: 80%">
               <td valign="top" align="left">
                   <span style="font-family:verdana,arial,
                                sans-serif;
                                font-size:11px;">Price:</span>
               </td>
               <td valign="top">
                  <span class="itemNameText" >$
                  <span dojoAttachPoint="priceElement">
                  </span></span>
               </td>
               <td valign="top" align="right">
                  <button dojoType="dijit.form.Button"
                   dojoAttachEvent="onclick: addToCart">
                   Add to cart</button>
               </td>
            </tr>
          </table>
      . . . . .
      . . . . .
      // Additional content
      . . . . .
      . . . . .
      </table>
</div>
```

The widget's template and CSS defines the appearance of the widget and identify where in the DOM updates will occur. The next step is to add the JavaScript code to the widget that does the work.

Listing 4.0 shows the beginning section of the customized ItemDetail widget. The code begins by declaring the widget to be created: dojo.provide("ibm.widget.ItemDetails");

The name is used to reference the newly created widget.

Below the dojo.provide is the dojo.declare. The dojo.declare defines the widget and any inheritance that this widget might have. In this case, the code declares inheritance from base dijit._Widget and dijit._Templated. Since this is a templated widget, the code declares the templatePath. The template path defines the location of the template that was defined above. The dojo.moduleUrl is a utility function that is used to resolve the location of the template.

Further down in the code, the DojoAtachPoint declarations defined in listing 3.0 are declared. These will be the root nodes of the DOM references that will be updated. In this case nameElement, descElement are declared among a number of other variables. The article will discuss in the next section how these elements are modified.

Listing 4.0: Declaring the ItemDetails widget

```
dojo.provide("ibm.widget.ItemDetails");

dojo.declare(
    "ibm.widget.ItemDetails",
    [dijit._Widget, dijit._Templated],
    {
      initializer: function(){ },
      templatePath:
      dojo.moduleUrl("ibm","widget/templates/ItemDetails.html"),
      isContainer: false,

      // your DOM nodes declared as DojoAttachPoint in the
      // Template:
      nameElement: null,
      descElement: null,
      imageElement: null,
      priceElement: null,

     // Additional Code
     . . . .
     . . . .
```

To really make the widget interesting, data is needed to populate the widget. The data for the details widget is located on the server. How does one get the description data to render in the page?

The Dojo Toolkit offers a powerful IO Transport mechanism that can be used to issue XMLHttpRequests (XHR) requests to the server and process the result. The XHR API opens an independent communication channel within the browser. XHR is at the heart of what gives Ajax its interactive feel.

Listing 5.0 shows the use of the dojo.xhrGet function to retrieve data from the server. The Dojo Toolkit wrappers the XHR API with it's own API which makes dealing with the XHR API easier. The *url*: defines the address of the server. The ?p0="+item.id is a URL parameter that is passed to the server. The *timeout* defines how long to wait before giving up on the connection to the server. The *headers* define additional HTTP headers that should be applied to the request sent to the server. The *handleAs* tells the Dojo Toolkit how to handle the response. In this case, the expected response is XML code.

The deferred.addCallback(**function**(response) is the callback function that will be invoked by the Dojo Toolkit when the response is returned from the server. The previous section covered the DojoAttachPoint as being the location of DOM node to be modified. The descElement is one of the DOM nodes modified here. The self.**descElement**.innerHTML value is set to the description text that was returned in the XML. The descElement is the DOM node that was declared previously in the Template file of the widget. See listing 3.0.

Listing 5.0: XHR GET request to the server

```
var deferred = dojo.xhrGet( {
                      url:      self.url+"?p0="+item.id,
                      timeout:  5000,
                      headers:  { "Content-Type":"text/html" },
                      handleAs: "xml"
                  } );

deferred.addCallback(function(response){

     var itemElement =
         response.getElementsByTagName( "entry" )[0];

     self.descElement.innerHTML =
itemElement.getElementsByTagName( "description" )[0].firstChild.nod
eValue;

     self.hiddenData.setAttribute("itemname",  item.name);
     self.hiddenData.setAttribute("itemprice", item.price);
     self.hiddenData.setAttribute("itemid",    item.id);

     . . . . .

     return response;
   });
```

Once your customized Dojo Widget is created, it needs to be inserted into your HTML page so it can be used. Figure 8.0 shows what the page looks like. The listing is from the flowers.html file. Listing 6.0 shows a fragment of the HTML behind the page. The DIV tag is used to embed the widget and contains the keyword *dojoType* which declares

the ItemDetail widget. There are additional parameters passed to the widget in the form of *url* which is the URL request to the RPCAdapter on the server. If you recall, the ItemDetails widget makes an XHR request to the server to retrieve its data to display. The server side code will be covered in the next section which looks at the RPCAdapter usage.

Listing 6.0: Declaring ItemDetails in the HTML page

```
<body style="background: #FFF;">
   . . . .
   . . . . //additional HTML
   . . . .

<div dojoType="ibm.widget.ItemDetails"
url="/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample/detai
lRequest"
style="width: 100%;"
itemTopic="itemdetails_flowers"></div>

</body>
```

Looking at the Server side.

At this point, you have seen how a customized widget is created in Dojo and how that widget would go about issuing XHR request to the server. Let's look at how the server might handle those requests.

In the case of the Plants By WebSphere application, the widget issues an XHR request to the server for some data. The Servlet reads the request, looks up the correct EJB. The EJB is called and the EJB container issues a request to the database which returns the result. The result needs to be encoded into XML and returned back to the JavaScript widget on the browser.

For this scenario, the Remote Procedure Call Adapter (RPCAdapter) which is provided with IBM feature pack for WebSphere Application Server will be used to connect to the Plants By WebSphere EJBs

The RPCAdapter runtime JAR is included with the Plants By WebSphere application and is configured using an XML file located in the WEB-INF of the WAR file. Listing 7.0 shows an example configuration.

Listing 7.0 – RpcAdapterConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rpcAdapter>
   <default-format>xml</default-format>
   <services>
       <pojo>
          <name>Sample</name>

<implementation>
   com.ibm.websphere.samples.plantsbywebspherewar.RpcConnecter
</implementation>

          <methods filter="whitelisting">
            <method>
               <name>detailRequest</name>
               <description>
                     returns back detail information for an item
               </description>
               <http-method>GET</http-method>
                   <parameters>
                        <parameter>
                            <name source="request">message</name>
                            <description>
                               Contains the message to be returned
                            </description>
                        </parameter>
                   </parameters>
            </method>
          </methods>
       </pojo>

   </services>
</rpcAdapter>
```

Let's look closer at the RpcAdapterConfig.xml file. The <default-format> returned by the RPCAdapter is declared as XML. The adapter also supports returning data in JavaScript Object Notation (JSON) as well. The <implementation> element contains the user defined class definition the RPCAdapter will instantiate to invoke the method. The method name in the class to call is defined in the <name> element and is called detailRequest. The <http-method> the RPCAdapter is expecting is the GET. The parameter that is expected on the URL request is called 'message'.

Putting it all together, the ItemDetail widget from Figure 8.0 would invoke the RPCAdapter using the following format where F001 is the identifier of the item.

```
/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample/detailRequest?message=F0001
```

The result returned by the RPCAdapter would be XML data. Figure 9.0 shows the XML output.

Figure 9.0: XML output returned to the ItemDetail widget from the RPCAdapter

```
<results>
     <entry>
      <description>
African orchids are some of the most endangered and rare kinds of
orchids grown today. This variety is medium yellow with varigated
salmon and pink insides. Height: 18 to 28 inches.
           </description>
           <image>
/PlantsByWebSphereAjax/servlet/ImageServlet?getimagebyid=F0001
           </image>
           <id>F0001</id>
           <name>African Orchid</name>
            <price>$250.00</price>

           <thumb>
/PlantsByWebSphereAjax/servlet/ImageServlet?getimagebyid=F0001
           </thumb>
           <dept>0</dept>
           <heading>Rare Delicate Beauty</heading>
      </entry>
</results>
```

Lets take a closer look a the implementation class. This was defined in the RpcAdapterConfig.xml file and contains the detailRequest method. Listing 8.0 shows

the implementation class com.ibm.websphere.samples.plantsbywebspherewar.RpcConnecter defined in the RpcAdapterConfig.xml configuration file in the previous listing.

Listing 8.0 – implementation class com.ibm.websphere.samples.plantsbywebspherewar.RpcConnecter

```
public Collection detailRequest( String itemId )  {

   ItemDetailPojo itemDetailPojo = new ItemDetailPojo();
   Vector itemCollection = new Vector();

   if (itemId != null) {
      try
      {
         Catalog catalog = catalogHome.create();
         StoreItem item = (StoreItem)
                          catalog.getItem(itemId);

         if ( item != null ) {
            itemDetailPojo.name        = item.getName();
            itemDetailPojo.id          = item.getID();
            itemDetailPojo.price       =
java.text.NumberFormat.getCurrencyInstance(java.util.Locale.US).for
mat(new Float(item.getPrice()));

            itemDetailPojo.heading     = item.getHeading();
            itemDetailPojo.description = item.getDescription();
            itemDetailPojo.dept        = new
                           Integer(item.getCategory()).toString();
            itemDetailPojo.thumb       =
"/PlantsByWebSphereAjax/servlet/ImageServlet?getimagebyid="+item.ge
tID();
            itemDetailPojo.image       =
"/PlantsByWebSphereAjax/servlet/ImageServlet?getimagebyid="+item.ge
tID();

             itemCollection.add(itemDetailPojo);
         }

       }
            catch (javax.ejb.CreateException e)
            {
                Util.debug("RpcConnecter: EJB Create Exception in
                           detailRequest(...)");
            }
            catch (Exception e)
            {
                Util.debug("RpcConnecter: general Exception in
                           detailRequest(...)");
            }
    }
    return itemCollection;
}
```

The method detailrequest( ) returns back a Collection of POJO elements. The Collection class is part of the java.util package. The POJO elements are derived from data returned by invoking the StoreItem EJB. The StoreItem is retrieved from the Catalog EJB. The Catalog EJB is a collection of the items that are contained within the Plants By WebSphere inventory located in the Derby database. Once a Java Collection is returned, the RPCAdapter will transparently map the Collection to XML data. The Collection's key corresponds to the XML element and the key's value to the XML value. The XML stream is returned to the ItemDetail widget.

By using the RPCAdapter and having it return XML, you have in affect created a service that others can connect too as well. The service only returns data and it would be up to the caller to figure out how to render the data. A usage scenario might be a partner company of Plants By WebSphere that aggregates data from its own greenhouse with the catalog information maintained by Plants By WebSphere. This scenario is sometimes referred to as a mashup.

Other Scenarios to Think About

While the Plants By WebSphere application does not implement the specific scenarios listed below, they are nevertheless helpful in understanding other ways to use the benefits of the IBM WebSphere Application Server Feature Pack for Web 2.0 in J2EE applications.

Proxy services

The article talked briefly about creating a mashup application, combining content from multiple sites to create the appearance of one site. The classic mashup example is taking Google maps and customizing it with user unique location based content. One of the challenges in creating mashups is dealing with cross site scripting within the browser. As an example, if you go to mydomain.com to access the Plants By WebSphere application, but the widgets you created issue XML requests to mypartner.com using XHR, then the browser will prevent the request. Normally this is a very good behavior since it prevents cross-site scripting vulnerabilities from occurring when you access pages on the Web.

In the case of Plants By WebSphere, you want to allow your widgets to access other sites for content. How do you allow cross site scripting responsibly within the browser?

One common way is to use a forward proxy. A forward proxy takes the request from the browser, looks at the URL, and forwards the requests to the appropriate domain. From the browsers perspective, the information appears to come from the same domain, but in reality, the content request is made by the proxy on behalf of the client. The Ajax proxy that is shipped with the IBM WebSphere Application Server Feature Pack for Web 2.0 includes a customizable Servlet proxy.

The proxy is Servlet based and can be embedded directly in an EAR file or run on your server as a WAR file. The proxy also includes a white-listing ability which you can further use to restrict the kinds of requests that the proxy is allowed to service. The proxy also includes the ability to filter on headers, mime-types, and cookies. The proxy is customizable using an XML file.

Feed Syndication – tell the world
Web syndication is making available content to other sites to use and is often called Web feeds. Typically, a site makes available a Web feed which contains a title and a short description of the content. If the user is interested in the content, they click on the link and are taken to the site to read more detailed information. In the case of the Plants By WebSphere application, one can imagine a scenario where a Web feed contains additional planting tips, sales, or blogs by horticulture specialists.

The IBM Feature pack for Web 2.0 provides the Apache Abdera library which includes an implementation of the Atom Syndication Format and Atom Publishing protocol to help you develop your own feeds.

A syndication solution would not be complete without taking into account the client side implementation. The IBM feature pack provides as part of the Dojo Toolkit extensions an Atom Feed viewer. The viewer can be embedded in your HTML as a widget to provide feed viewing capabilities.

Resources

- *IBM® WebSphere® Application Server Version 6.x Feature Pack for Web 2.0*
  http://www-306.ibm.com/software/webservers/appserv/was/featurepacks/web20/

- Dojo Toolkit Web site.
  http://www.dojotoolkit.org

- Dojo Toolkit documentation
  http://www.dojotoolkit.org/docs

- Apache Abdera
  http://incubator.apache.org/abdera/

About the Author

Kevin Haverlock is an architect and developer with IBM's WebSphere Application Server product. He is currently part of IBM's WebSphere Application Server Feature Pack for Web 2.0 team. Kevin can be reached at kbh@us.ibm.com
.