www.phparch.com

php | architect™

The Magazine For PHP Professionals

planet http://

GOING BEYOND THE BROWSER

PHP-GTK 2

Using Glade to quickly and easily design desktop applications

INTEGRATING PHP WITH OpenLaszlo®

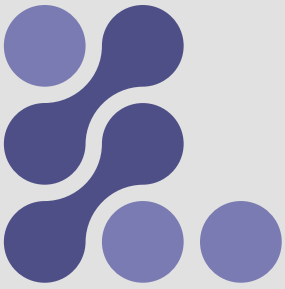
DON'T REPEAT YOURSELF

Staying DRY with code generation in PHP

VOLUME 5 ISSUE 8

SECURITY CORNER:
PHP CONFIGURATION SECURITY

TEST PATTERN:
VARIATIONS ON A THEME



OpenLaszlo®

Integrating with PHP

Rich Internet applications are a hot topic at present. This article introduces OpenLaszlo, an upcoming technology designed to address this issue using XML and Flash, and explains how to embed OpenLaszlo applications into your PHP scripts—and how you can use PHP as a bridge to communicate MySQL data via XML.

by **TITUS BARIK**

PHP: *5*
O/S: *Debian Linux*
OTHER SOFTWARE: *Apache 2, Sun JDK 1.5, OpenLaszlo*

LINKS:
http://www.openlaszlo.org/
http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

CODE DIRECTORY: *laszlo*

TO DISCUSS THIS ARTICLE VISIT:
http://forum.phparch.com/321

There is a real need for a rich Internet application platform that conventional Web technologies fail to adequately provide. The Web, for the majority of its history, has been essentially a document-based platform. As a result, current approaches such as DHTML, JavaScript and—more recently—AJAX, have all attempted to transform this document-centric model into an application-centric model. However, these technologies fall short of filling the needs of an application platform because they are, at their core, simply extensions of an already outdated architecture.

OpenLaszlo is an open source platform that offers a media-rich Internet application alternative. Laszlo is an architecture built from the ground up to support application-centric development and deployment. Although it's difficult to describe the abilities of OpenLaszlo directly in a text medium, the highly accessible Pandora music service available at `http://pandora.com` demonstrates the rich capabilities of the OpenLaszlo architecture in practice; the Pandora music service uses OpenLaszlo as its application platform. If you don't have Internet access handy right now, you can see what Pandora looks like in Figure 1.

As there is already extensive documentation and promotional literature on working and developing applications directly within the OpenLaszlo architecture, an article dedicated to such a topic would be redundant. Similarly, this article is not a tutorial on LZX, Laszlo's XML-based programming language. On the project Web site you can find the source code, a selection of examples and a quick start guide—as well as the language documentation—to fill this role. The focus of this article rather targets Laszlo or PHP developers who wish to harness the capabilities of the OpenLaszlo

OpenLaszlo provides several API methods for communicating with XML.

front-end from within a PHP back-end environment. It covers the configuration, installation, OpenLaszlo language constructs, and PHP coding practices that will enable you to effectively utilize Laszlo and make the most of this upcoming Web architecture from within your applications.

OpenLaszlo Architecture Overview

OpenLaszlo offers a variety of configuration options that affect the way the platform is built, and subsequently deployed. This section will focus exclusively on one of the more complex, but flexible, configurations—proxied mode.

At the highest architectural level, OpenLaszlo can be deployed in one of two ways: as `proxied mode`, or as `SOLO mode`. The `proxied mode` configuration option we're interested in here requires a J2EE server to compile the requested Laszlo source programs (`LZX` files) and present the resulting binary to the Web client. In the OpenLaszlo Developer Kit, Apache Tomcat is included as the J2EE server.

To go into more detail: when a client makes a request for a Laszlo application in `proxied mode`, the request is proxied on to the J2EE server and compiled into Macromedia Flash byte code using the Laszlo Presentation Server (LPS). It's then returned to the caller as a Flash document. The Laszlo Presentation Server consolidates the roles of compiling the source XML program, transcoding media formats, connecting to third-party data sources `(through XML services)`, and caching Laszlo applications. In a way, Macromedia Flash acts as a virtual machine for OpenLaszlo.

In our applications, Laszlo is not running in isolation, and must interface with our PHP code. In this situation, Apache 2 needs to be configured as the primary web server—running PHP of course—and any content that requires the use of LPS is passed, or proxied, to the secondary Tomcat server through the use of connector and adaptor modules. At the first take, such a setup may seem peculiar, but it is in fact a fairly standard configuration in real-world situations where multiple language support and/or advanced configuration are necessary. When properly configured, this setup also allows for greater security. Apache can filter requests and pass them on in a limited manner as necessary to Tomcat, rather than having Tomcat be directly accessible to the Web browser. Figure 2 depicts the components and interactions for such a setup.

Basically, Apache 2 proxies requests to Tomcat, which in turn invokes the Laszlo Presentation Server, which then returns the compiled Flash SWF file to the caller.

Installing OpenLaszlo

In this section, I will describe the steps required

FIGURE 1

to install OpenLaszlo. For more specific installation instructions, you might want to refer to the OpenLaszlo documentation.

Although configuring the Apache server to communicate with LPS has its nuances, the installation of the OpenLaszlo platform itself is simple. The bundled version of OpenLaszlo has only one dependency: it needs an installed Java SDK on the machine. Therefore, you'll need to ensure that Java is installed first. Don't forget to make sure that the `JAVA_HOME` environment variable is set to point to your Java installation! Once that's in place, if you're running Linux you will need to download the OpenLaszlo Development Kit tarball from the project web site and extract the contents into `/usr/local`. If you're on Windows or MacOSX, there is an installer available on the site that will do all the dirty work for you.

The next thing is to start up the Tomcat server. Under Windows there's a click-and-go button on the Start menu, `Start/Programs/OpenLaszlo Server/Start OpenLaszlo Server`; under MacOSX there's an even more straightforward desktop icon, `OpenLaszlo Start Tomcat.term`, which will start Tomcat with a double click. Under Linux, you will need to run Tomcat's startup shell script:

```
./lps-3.3.1/Server/tomcat-5.0.24/bin/startup.sh
```

Regardless of the underlying operating system, the Tomcat server will run on port `8080`. Also regardless of the operating system, OpenLaszlo's "Hello World" test application can be accessed through the following URL:

```
http://example.org:8080/lps-3.3.1/examples/hello.lzx
```

Assuming all went well with your installation, the text "Hello Laszlo!" should appear in your Web browser. You can right-click on the text to verify that the text genuinely originates from a compiled Flash SWF file.

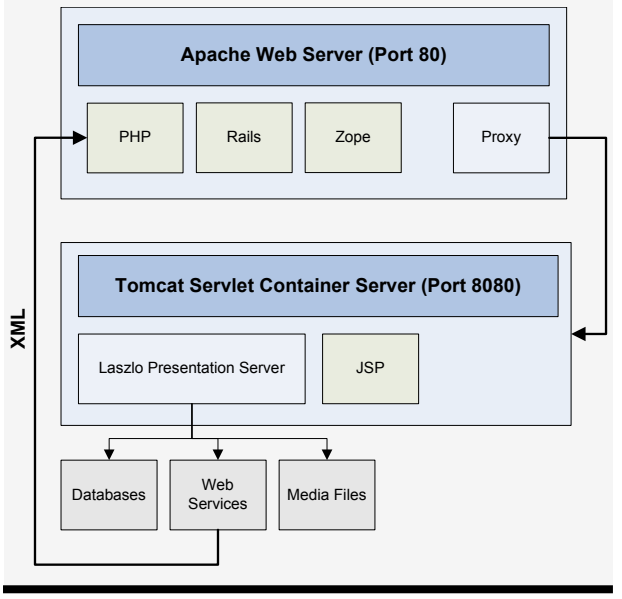
Below the text label, you should see a development console. This console appears when you're working with OpenLaszlo in development mode. It allows you to experiment with the currently loaded application, and should look something like the screen shot in Figure 3. Having a console specifically tailored to development needs enables you to debug your application and test different deployment configurations.

If you would like a complete listing of the examples and documentation provided with the OpenLaszlo bundle, point your browser to:

```
http://example.org:8080/lps-3.3.1
```

It is worth spending a few moments exploring the same applications and their source code to get a feel for the way OpenLaszlo operates. For example, Figure 4 displays a Web-based calendar—one of the many rich applications written in Laszlo. This example calendar can be updated and/or read from any machine; no layout or graphic content is requested from the server, just data, pure and simple.

FIGURE 2



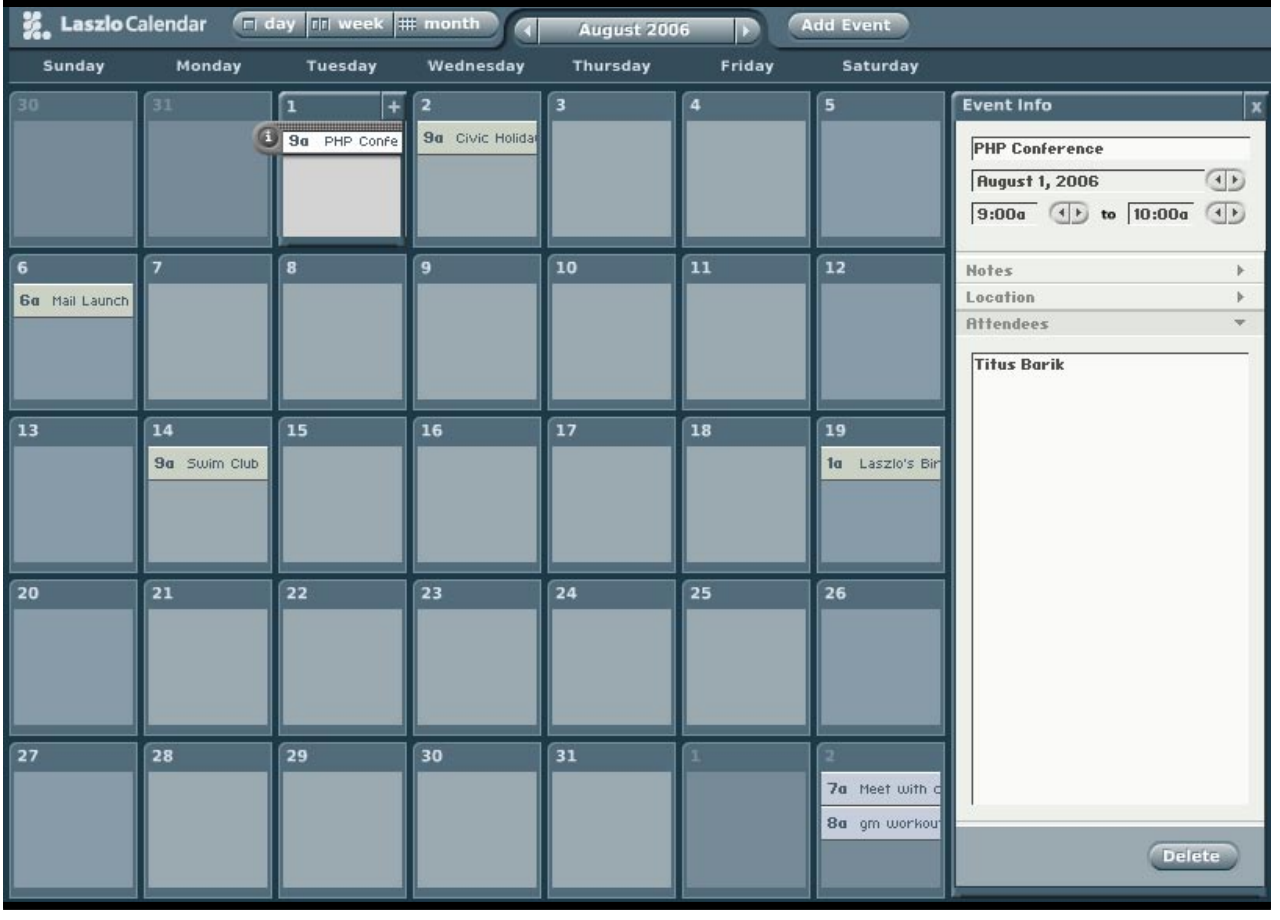
Connecting to Apache 2

For the purposes of this article, I will assume that you have an existing, working Apache 2 setup with PHP 5 installed on it, and that you are comfortable with making changes to your Apache configuration.

There are a variety of adapters available to connect Apache and Tomcat, including `mod_jk`, `mod_jk2`, and `mod_jserv`. My recommendation is to ignore all these server-specific connectors, unless you have a very particular reason to use them, and use the more flexible `mod_proxy` module that is provided with the Apache distribution.

FIGURE 3

FIGURE 4



To enable the proxy module under Linux, if you have `a2enmod` on board you can simply type:

```
a2enmod proxy
```

This command enables `mod_cache`, `mod_disk_cache`, `mod_proxy`, and most importantly, `mod_proxy_http`. If you aren't using Linux or don't have the `a2enmod` command, you will need to add these modules to Apache's `httpd.conf` manually using the `LoadModule` directive.

Next, you will need to modify the Proxy section of `httpd.conf` (or `proxy.conf` if you're using that) to allow clients to connect:

```
ProxyRequests Off

<Proxy *>
Order deny, allow
Allow from all
</Proxy>
```

Modify the desired site (usually default) under the sites-available directory to proxy requests under the `/lzx` directory to Tomcat:

```
ProxyPass /lzx http://localhost:8080/lps-3.3.1
ProxyPassReverse /lzx http://localhost:8080/lps-3.3.1
```

This configures a proxy and a reverse proxy, respectively. Apache's `ProxyPass` directive allows remote servers to be mapped into the space of the local server; while the `ProxyPassReverse` directive implements the reciprocal operation and causes the response location to return in the correct form. The use of `localhost` in the third argument implies that one cannot generally connect to Tomcat, but must do so over a public connection through Apache, assuming that Apache and Tomcat are running on the same machine.

For development purposes only, you'll also want to add `lps-3.3.1` as a proxy directory, as many of the paths in OpenLaszlo are hard-coded to look at this specific location.

```
ProxyPass /lps-3.3.1 http://localhost:8080/lps-3.3.1
ProxyPassReverse /lps-3.3.1 http://localhost:8080/lps-3.3.1
```

OpenLaszlo is an open source platform that offers a media-rich Internet application alternative.

Finally, you will need to make a small modification to the Tomcat configuration file, `server.conf`. Change the `Connection` parameter to:

```
<Connector
            port="8080"
            proxyName="example.org"
            proxyPort="80" />
```

This will cause servlets inside the Tomcat Web application to think that all proxied requests were directed to `example.org` on port `80`.

You can test whether or not the proxy is functioning correctly by accessing Tomcat from within the `lzx` URL structure:

```
http://example.org/lzx/examples/music/music.lzx
```

This URL will load the multimedia music application as shown in Figure 5. If you've reached this far, your configuration of Apache and Tomcat are complete.

The small application, `Music`, streams an `mp3` file over HTTP. OpenLaszlo provides functions to manipulate the media stream.

Embedding Laszlo From Within PHP

So far, I've demonstrated several standalone Laszlo applications, but none of these demo applications have run as components embedded within other content on a Web page. There are, in fact, several ways to embedded Laszlo applications within a standard HTML page, depending on your requirements. OpenLaszlo even provides convenience methods, through JavaScript, that facilitate the insertion of such components into HTML. This is done by referencing a Laszlo application and passing to it an `lzt` target parameter.

For example, if the target platform supports JavaScript, one can add the `embed.js` JavaScript file to the HTML `<head>` section:

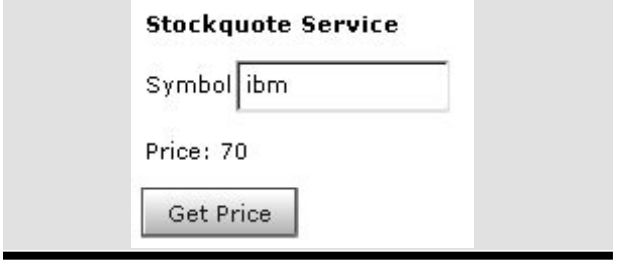
```
<script src="/lzx/lps/includes/embed.js"
  type="text/javascript">
</script>
```

FIGURE 5

FIGURE 6

FIGURE 7

FIGURE 8



There are several ways to embed Laszlo applications within a standard HTML page, depending on your requirements.

This in turn enables the `lzEmbed` JavaScript function, allowing you to embed content with your pages. You can now tell the OpenLaszlo application to simply return the appropriate JavaScript, by specifying an `lzt` target of `js`:

```
<script
  src="/lzx/demos/vacation-survey/vacation-survey.lzx?lzt=js"
  type="text/javascript">
</script>
```

If you need more flexibility—for example, say, you want to set the background color—you can manually add the `lzEmbed` function, like so:

```
<script type="text/javascript">
lzEmbed
{
  url: '/lzx/demos/vacation-survey/vacation-survey.lzx?lzt=swf',
  bgcolor: '#ffffff',
  width: '600',
  height: '206'
}
</script>
```

Alternatively, if JavaScript is not available, a standard HTML `object` tag can be used:

```
<object type="application/x-shockwave-flash"
        data="vacation-survey.lzx?lzt=swf&debug=false&lzr=swf7"
        width="600" height="206">
  <param name="movie" value="vacation-survey.lzx?lzt=swf&debug=fals
e&lzr=swf7" />
  <param name="quality" value="high" />
  <param name="scale" value="noscale" />
  <param name="salign" value="LT" />
  <param name="menu" value="false" />
</object>
```

To aid in embedding, OpenLaszlo provides the `Server` and `SOLO` options via the development console that was alluded to earlier in this article. You can simply click on these buttons to generate the appropriate XHTML code, which you can then copy and paste into your web page.

By default, having no `lzt` parameter places the application in development mode, where you can view the source code, compile for different available targets, and debug the application. In most cases, these features are not desirable on a production server. To enable and disable `lzt` request types on a case by case basis, edit the `lps.properties` file found in `WEB-INF/lps/config`, under `Server/lps-3.3.1`. For example, to disable viewing of LZX source code, set the `allowRequestSOURCE` parameter to `false`. To change the default request type, set the `defaultRequestType` parameter.

Figure 6 demonstrates the stock vacation survey application running in non-development mode from within a PHP page, using the `embed.js` library. Above and below the Laszlo application is a set of `<h1>` tags to illustrate that it is indeed embedded within HTML content. Note that the development console does not appear because the `lzt` target type is `swf`—the Laszlo Vacation application is embedded within the content of a standard PHP or HTML page.

OpenLaszlo and XML

OpenLaszlo is based on XML, and it is therefore unsurprising that its primary interface with external data is also XML. OpenLaszlo provides several API methods for communicating with XML. These include RPC services, such as SOAP and XML-RPC, as well as databinding through datasets, datapaths, and XPath queries. In the following sections, we will use SOAP and REST-like services to pull and push data to PHP. This flexibility enables us to communicate with a variety of data sources, as long as they can "speak" XML back to Laszlo.

Hello OpenLaszlo

As mentioned previously, this article is not a tutorial on OpenLaszlo programming. However, a simple "Hello

LISTING 1

```
<canvas>

 <dataset name="ds">
  <colors>
   <color>Blue</color>
   <color>Red</color>
   <color>Green</color>
   <color>Yellow</color>
  </colors>
 </dataset>

 <window width="200" height="150" y="10">
 <view>
  <simplelayout axis="y" />
  <text id="mytext" width="200">Hello World!</text>

  <button text="Click Me">
   <handler name="onclick">
    mytext.setText("Goodbye World!");
   </handler>
  </button>

 </view>
 </window>

 <window width="200" height="200" x="250" y="10">
  <view datapath="ds:/colors">
   <simplelayout axis="y" />
   <text datapath="color/text()" />
  </view>
 </window>

</canvas>
```

LISTING 2

```
<?php

$quotes = array(
        "amd" => 15.50,
        "ibm" => 70,
        "msft" => 20.80
);

function getQuote($symbol) {
    global $quotes;
    return $quotes[$symbol];
}

ini_set("soap.wsdl_cache_enabled", "0"); // disabling WSDL cache
$server = new SoapServer("stockquote.wsdl");
$server->addFunction("getQuote");
$server->handle();

?>
```

LISTING 3

```
<?xml version='1.0' encoding ='UTF-8' ?>
<definitions name='StockQuote'
 targetNamespace='http://example.org/StockQuote'
 xmlns:tns=' http://example.org/StockQuote '
 xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
 xmlns='http://schemas.xmlsoap.org/wsdl/'>

 <message name='getQuoteRequest'>
  <part name='symbol' type='xsd:string'/>
 </message>
 <message name='getQuoteResponse'>
  <part name='Result' type='xsd:float'/>
 </message>

 <portType name='StockQuotePortType'>
  <operation name='getQuote'>
   <input message='tns:getQuoteRequest'/>
   <output message='tns:getQuoteResponse'/>
  </operation>
 </portType>

 <binding name='StockQuoteBinding' type='tns:StockQuotePortType'>
  <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
   <operation name='getQuote'>
   <soap:operation soapAction='urn:xmethods-delayed-quotes#getQuote'/>
   <input>
    <soap:body use='encoded' namespace='urn:xmethods-delayed-quotes'
     encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
   </input>
   <output>
    <soap:body use='encoded' namespace='urn:xmethods-delayed-quotes'
     encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
   </output>
  </operation>
 </binding>

 <service name='StockQuoteService'>
  <port name='StockQuotePort' binding='StockQuoteBinding'>
   <soap:address location='soapserver.php'/>
  </port>
 </service>
</definitions>
```

World" application serves to illustrate the code that comprises a typical LZX source file, shown in Listing 1. Figure 7 shows the compiled source code given in Listing 1, in action. While the example is artificial, it illustrates a few basic points regarding the way LZX source code is structured.

The base of any Laszlo source file is the `canvas` tag. The `canvas` element represents the boundaries of a Laszlo application. Within this particular canvas, there are two windows and an embedded dataset. The first of these windows displays the text "`Hello world`." A button with an `onclick` event handler serves as a trigger to change this text to "`Goodbye world`" when clicked.

The Hello World application demonstrates the XML source code for placing components, scripting components, and working with XPath and datasets.

The second window populates a set of text labels using the embedded dataset. The `datapath` attribute uses a simple XPath query to pull data from a dataset. Because the view has a `datapath`, and because the referenced path is not a single element, the view iterates over the text element until all data in the dataset is exhausted.

If the dataset had instead been stored in a static, external XML file, one might change the dataset line to the following:

```
<dataset name="ds" src="colors.xml" />
```

If, on the other hand, the XML file was dynamic—say, created by a generated PHP page—then one would use the type attribute, with a request attribute indicating that the dataset should be loaded on application startup:

```
<dataset name="ds" src="colors.php"
  type="http" request="true" />
```

If you've programmed before in XUL or XAML, you'll feel right at home with LZX. At this point, you should have a rough idea as to the ways in which OpenLaszlo will interact with PHP using XML. It's now time to explore these interactions in detail.

Talking to PHP with SOAP

Whereas previously I showed you how to communicate with OpenLaszlo through PHP passively, this section actively uses Simple Access Object Protocol (SOAP) libraries to exchange data with PHP. Unfortunately, unlike SOAP technologies in J2EE, PHP SOAP implementations have not yet fully matured. Still, I use the native PHP 5 SOAP functions, though they do not intrinsically provide the ability to auto-generate WSDL files at this moment in time.

In this SOAP example, I implemented a SOAP server that provides a fake "Delayed Stock Quote" service that mimics the behavior of the XMethods demo service. The example is taken from by the Zend article "PHP Soap Extension," by Dmitry Stogov (http://www.zend.com/php5/articles/php5-SOAP.php), and is expanded on to interface with OpenLaszlo. The SOAP Server source code is shown in Listing 2. It provides a `getQuote` method with hard-coded data that is made accessible through SOAP via the PHP SoapServer `addFunction()` function. The corresponding WSDL file, `stockquote.wsdl`, is written by hand, and shown in Listing 3. Remember to modify the `soap:address` line of the WSDL at the bottom of the file to point to your PHP SOAP server location.

The code in the LZX file in Listing 3 uses the XML `soap` element to interface with our Delayed Stock Quote service. Of particular interest are the `onload` handler, which ensures that the presentation client is able to successfully connect to the SOAP service, and the `remotecall` element, which specifies the available SOAP functions. An `onclick` event handler is associated with the `request price` button, which in turn invokes the SOAP function through JavaScript. The global `ondata` handler then updates the text label with the stock price. If the SOAP service had more than one callable function, then a custom `ondata` handler could be provided within each `remotecall` element. You can see a screenshot of the Stockquote Service application in action in Figure 8.

A SOAP service may not necessarily be available or feasible to implement, depending on your PHP application. I'll now show you an alternative mechanism to interface with PHP, through the use of datasets and standard HTTP `GET` and `POST` operators.

Talking to PHP through GET and POST

In this section, I will show you how to use REST-like protocols (`POST` and `GET`) to read and write data to a MySQL database from OpenLaszlo. Recall that OpenLaszlo works with XML datasets, and not directly with database servers. PHP is therefore used as a bridge that connects to the database, and returns the result set in an XML format appropriate for Laszlo.

Listing 5 contains the source code for a sample contact manager application. It is designed to closely match the sample contact manager provided with OpenLaszlo, but uses PHP rather than JSP as the server-side XML generator. The `getcontacts.php` file, shown in Listing 6, simply performs standard MySQL calls and wraps the results in XML. The OpenLaszlo application retrieves data from this PHP page through the `dset` dataset defined in the LZX file.

LISTING 4

```
<canvas debug="true">

 <soap name="maths" wsdl="stockquote.wsdl">

  <handler name="onload">
   canvas.buttons.setAttribute('visible', true);
  </handler>

  <remotecall funcname="getQuote">
   <param value="${a.text}" />
  </remotecall>

  <handler name="ondata" args="value">
   result.setText(value);
  </handler>

 </soap>

 <view name="buttons" x="10" y="10" visible="false" layout="spacing: 10" >
  <text><b>Stockquote Service</b></text>

  <view layout="axis: x">
   <text y="3">Symbol</text><edittext id="a" text="ibm"/>
  </view>

  <view layout="axis: x">
   <text>Price:</text><text id="result"/>
  </view>

  <button text="Get Price"  onclick="canvas.maths.getQuote.invoke()" />
 </view>

</canvas>
```

LISTING 5

```
<canvas bgcolor="#D4D0C8">
 <dataset name="dset" src="getcontacts.php" request="true" type="http"/>
 <!-- 1 -->
 <dataset name="dsSendData" request="false" src="contactmgr.php" type="http"/>

 <class name="contactview" extends="view" visible="false" x="20" height="120">
  <!-- 2 -->
  <text name="pk" visible="false" datapath="@email"/>
  <text y="10">First Name:</text>
  <edittext name="firstName" datapath="@firstName" x="80" y="10"/>
  <text y="35">Last Name:</text>
  <edittext name="lastName" datapath="@lastName" x="80" y="35"/>
  <text y="60">Phone:</text>
  <edittext name="phone" datapath="@phone" x="80" y="60"/>
  <text y="85">Email:</text>
  <edittext name="email" datapath="@email" x="80" y="85"/>
  <method name="sendData" args="action">
   var d=canvas.datasets.dsSendData;
   var p=new LzParam();
   p.addValue("action", action, true);
   p.addValue("pk", pk.getText(), true);
   p.addValue("firstName", firstName.getText(), true);
   p.addValue("lastName", lastName.getText(), true);
   p.addValue("phone", phone.getText(), true);
   p.addValue("email", email.getText(), true);
   d.setQueryString(p);
   d.doRequest();
  </method>
  <!-- 4 -->
 </class>

 <simplelayout axis="y"/>

 <view>
  <simplelayout axis="y"/>
  <text onclick="parent.newContact.setVisible(!parent.newContact.visible);">New En-
try...</text>
  <contactview name="newContact" datapath="new:/contact">
   <button width="80" x="200" y="10">Add
    <handler name="onclick">
     parent.sendData("insert");
     parent.datapath.updateData();
     var dp=canvas.datasets.dset.getPointer();
     dp.selectChild();
     dp.addNodeFromPointer( parent.datapath );
     parent.setVisible(false);
     parent.setDatapath("new:/contact");
    </handler>
   </button>
  </contactview>
 </view>

 <view datapath="dset:/phonebook/contact">
  <simplelayout axis="y"/>
  <view name="list" onclick="parent.updateContact.setVisible(!parent.updateContact.
visible);">
   <simplelayout axis="x"/>
   <text datapath="@firstName"/>
```

LISTING 5 (CONT'D)

```
    <text datapath="@lastName"/>
    <text datapath="@phone"/>
    <text datapath="@email"/>
   </view>
   <contactview name="updateContact">
    <button width="80" x="200" y="10">Update
     <handler name="onclick">
      parent.sendData("update");
      parent.parent.datapath.updateData();
     </handler>
    </button>
    <button width="80" x="200" y="40">Delete
     <handler name="onclick">
      parent.sendData("delete");
      parent.parent.datapath.deleteNode();
     </handler>
    </button>
   </contactview>
  </view>

</canvas>
```

LISTING 6

```
<phonebook>

<?php

mysql_connect('localhost', 'username', 'password');
mysql_select_db('db');
$result = mysql_query('SELECT * FROM contact');

while ($line = mysql_fetch_assoc($result)) {
    extract($line);
?>

 <contact firstName="<?=$first_name?>"
  lastName="<?=$last_name?>"
  phone="<?=$phone?>"
  email="<?=$email?>" />

<?php
}
?>

</phonebook>
```

LISTING 7

```
<?php

$action    = $_REQUEST['action'];
$firstName = $_REQUEST['firstName'];
$lastName  = $_REQUEST['lastName'];
$phone     = $_REQUEST['phone'];
$email     = $_REQUEST['email'];
$pk        = $_REQUEST['pk'];
$result    = null;

mysql_connect('localhost', 'username', 'password');
mysql_select_db('db');

switch ($action) {
    case 'insert':
        $sql = "INSERT INTO contact (first_name, last_name, phone, email) " .
           "VALUES (\"$firstName\", \"$lastName\", \"$phone\", \"$email\")";
        break;
    case 'update':
        $sql = "UPDATE contact SET first_name = '$firstName', ".
           "last_name = '$lastName', phone = '$phone', ".
           "email = '$email' WHERE email = '$pk'";
        break;
    case 'delete':
        $sql = "DELETE FROM contact WHERE email = '$pk'";
        break;
    default:
        $sql = null;
        break;
}

if ($sql) {
    $result = mysql_query($sql);
}

if ($result) {
    echo '<result>success</result>';
} else {
    echo '<result>failure</result>';
}

?>
```

The contactmgr.php file, in its entirety, is a dataset conforming to dsSendData—also defined in the LZX file—whose script is shown in Listing 7. Unlike dset, however, dsSendData is used as a write-only dataset, whose parameters are provided at run-time through scripting. The sendData() method constructs a set of parameters to send to the PHP script, and the doRequest() method invokes this request. Using datasets in such a manner allows you to pass data to PHP scripts, perform business logic, and return the results to your OpenLaszlo application.

By default, OpenLaszlo uses GET. One can convert a query to POST simply by setting the querytype parameter of the dataset to POST.

Conclusion

OpenLaszlo offers you the ability to generate rich presentation logic, without having to abandon the flexibility of the underlying PHP scripting. OpenLaszlo also provides a variety of components that are not available in traditional DHTML applications, and its native XML and XPath abilities simplify databinding to these components seamlessly.

That said, OpenLaszlo is not a perfect solution when it comes to embedded Web applications. The fact that it is based on Flash files can hinder accessibility and search engine ranking if used in excess, and while most platforms support Flash, a minority of browsers and users do not have this capability. Additionally, Macromedia offers its own rich commercial application environment: Flex. It remains to be seen whether Flex will hinder or help the adoption of OpenLaszlo. Plus, of course, AJAX continues to be a strong competitor to these technologies.

Despite these concerns, companies have already, and will continue to, implement applications using OpenLaszlo. If you find that your current AJAX application has grown in complexity, or if you've have hit a ceiling on its presentation capabilities, then you may want to give OpenLaszlo a try. ■

TITUS BARIK is a content application developer with an interest in open source Enterprise solutions. He has deployed both open source and proprietary content management systems successfully in corporate and non-profit environments. His personal weblog is available at barik.net, and he welcomes your comments and suggestions.