An Introduction to PAXI

next >

PHP Application XML Interface

- Built to support a variety of web services (XML) and to be deployed across any number of uniquely branded URLs
- Objective was to keep the framework extremely light weight and portable across many physical and virtual servers
- Client requirements were flexible templates & dynamic paramaters
- Personal requirements - no PEAR!
- Obviously wanted to use PHP for both speed and flexibility and its inherent template engine (see: Why PHP is a template engine?)

1 2 3 4 5 6 7 8 9 10 11 12 13 toc

Known Obstacles

- Web service used SOAP with attachments
- Web service didn't properly use SOAP protocol
- No information, except SOAP Fault, could be attained from the SOAP body or header (ie, couldn't continue process until XML document was parsed)
- PHP-SOAP, nuSOAP, PEAR though capable of building SOAP attachments do not currently support receiving/parsing SOAP with attachments
- Transport had to support POST over SSL
- cURL / PEAR complex implementation for POST over SSL
- XML files could be between 50k and 1.5Megs
- Needed XML values in an array to support dynamic templates (don't want to just transform the XML, ie, XSLT)
- At launch, framework needed to support 2k searches an hour - scaling to 10x that over three months
- **One month development timeline!**

The Solution

- Use sockets to connect over SSL to SOAP server
- Build custom SOAP client (pMime)
- Use SimpleXML to parse XML into an array (allowing access to data across dynamic templates)
- Decided not to use sessions to speed up development time
    - Used SOAP server's sessionID instead (since in most instances all user info is returned)
    - Allows for rapid scalability across multiple webservers

The Framework

- PHP 4 frontend validates user input, does a fuzzy match for airport codes
- PHP 4 frontend builds appropriate XML using buffers
- PHP 4 passes XML and sessionID (if appropriate) to PHP5 CLI
- PHP 5 CLI script (paxi.psh) communicates with PHP 4 Apache module via fast native UNIX pipes
- paxi.psh script determines request type and validates input
- Using SSL sockets, paxi.psh makes a SOAP request to remote server
- After validating the response and handling exceptions, parsed data is passed back into PHP 4 Apache module

PHP and Javascript, perfect together

< previous

next >

Dynamic JavaScript Tricks

City Name or Aiport Code:

Search!

- This function is called when you click the Submit button. If your browser is capable of it, a new <script> element will be appended to the element of the document, with the src attribute set to our airports.php script on the server.

```
function getAirports() {
  // call as popup for browsers that won't rescript on the fly
  if ( window.name != 'airports' && nodynamicjs ) {
    popupAirports();
  }
  else if ( nodynamicjs ) {
    //window.alert('submitting');
    document.getElementById('form1').submit();
  } else {
    // allow refresh by removing any previously appended script
    var aphead = document.getElementsByTagName('body').item(0);
    var apold  = document.getElementById('scriptId');
    if (apold) aphead.removeChild(apold);

    // create DOM script element
    newscript = document.createElement('script');
    var apfullpath = "http://example.com/airports.php?";

    // (snippet) get query values from form and add to scripturl
    if ( document.getElementById('destination1') ) {
          var dest1 = document.getElementById('destination1').value;
           apfullpath = apfullpath + 'destination1=' + destination1 + '&';
    }
    // assign src attribute to our script element
    newscript.setAttribute("src", apfullpath);
    // assign other attributes
    newscript.setAttribute("type",'text/javascript');
    newscript.setAttribute("defer", 'false');
    newscript.setAttribute("id", 'dynscript');
    newscript.setAttribute("version", '0.4');

    // append it to the head... nice trick (thanks D Kushner, DC Krook, J Knight)
    void(aphead.appendChild(newscript));  }
  }
```

The main processor function

- The following function takes a location query (like "St. Louis, MO") and a label (like "destination1"). It parses the query then checks to see if there are any airports or cities that match.
- If the the query is an airport code, TRUE is returned, indicating to the calling script that no choice needs to be made.
- If choices are found, a custom HTML <select> menu is returned listing each of the choices for that label.
- If nothing is found to match the query, an HTML message is returned requesting a different query.

```
// return (string) menu of Airports; or TRUE if valid Airport or City Code
function process($loc, $key) {
  // if $loc isn't already an airport...
  if ( !isAirport($loc) ) {
      // parse $loc for state/country names
      $loc_States = getStates($loc);
      // look up possible matches
       $loc_Airports = array();
       $loc_Choices = getAirports($loc_States, $loc_Airports);

            // if there are choices, render select menus
            if ( is_array($loc_Choices) ) {
                     $loc_menu = '<select class="dropdown"
                                    name="'.$key.'Select"
                                     onchange="document.getElementById(\''.$key.'\').value=this.value;" >
                                  <option value="">Please choose an airport...</option>';

                        foreach ( $loc_Choices AS $codearray ) {
                            $code = $codearray[0];
                             $citystate = $codearray[1];
                              $loc_menu .= '<option value="'.$code.'">'.htmlentities($citystate).'</option>';
                 }
                    $loc_menu .= '</select><span class="error">*</span>';
           }
          // or render message if no choices found
          else {
                      $loc_menu = '<div class="error">Aiport or City not found, please try again.</div>';
           }
  // quote the html for delivery
  $loc_menu = addslashes($loc_menu);
  }
  else {
       // loc is an airport code, proceed
          $loc_menu = TRUE;
  }
 return $loc_menu;
}
```

Returning the Javascript

- If all locations are valid airport codes, the following JavaScript is sent, which ensures that other form fields are valid, then submits the form:

```
if ( validateTripType(document.getElementById('form1'), $single) ) {
     document.getElementById('form1').submit();
}
```

- If not, we return JavaScript that renders the <select> menu of choices in the proper place on the from (destination1 in this case):

```
document.getElementById('destination1').innerHTML = "$destination1_menu";
```

The Basic Architecture

- PHP 4 frontend validates user input, does a fuzzy match for airport codes
- PHP 4 frontend builds appropriate XML using buffers

```
<?php

// PHP builds XML
$requestxml = buildXML ( $params );

// function with buffers to build XML
function buildXML ( $params ) {

ob_start();

print "<?xml version='1.0' encoding='iso-8859-1'?>";
?>
<nyphp xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="http://www.nyphp.org/add\Member.xsd">
  <memberID><?=$params['id']?></memberID>
  <firstName><?=$params['firstName']?></firstName>
  <lastName><?=$params['lastName']?></lastName>
  <? foreach ($params['array'] as $array) { ?>
    <list_info>
      <firstEl><?=$array[0]?></firstEl>
      <secondEl><?=$array[1]?></secondEl>
   </list_info>
  <? } ?>
</nyphp>
<?

return ob_get_clean();

}
?>
```

- PHP 4 passes XML and sessionID (if appropriate) to PHP5 CLI
- PHP 5 CLI script (paxi.psh) communicates with PHP 4 Apache module via fast native UNIX pipes
- paxi.psh script determines request type and validates input
- Using SSL sockets, paxi.psh makes a SOAP request to remote server
- After validating the response and handling exceptions, parsed data is passed back into PHP 4 Apache module

The Basic Architecture - cont

< previous

next >

- **PHP 4 to PHP 5 Communication**
    - PHP 4 validates form input from browser and generates SOAP packet using output buffering
    - Using **proc_open()** and command line arguments, PHP 4 controls and maintains bi-directional communication with paxi.psh

```
<?php

if( empty($sessionID) )
  $soap = proc_open(IPAXI_ARPSH,$fds,$soappipes);
else
  $soap = proc_open(IPAXI_ARPSH."{$sessionID}",$fds,$soappipes);

        fwrite($soappipes [0],$requestxml);

?>
```

1 2 3 4 5 6 7 8 9 10 11 12 13 toc

The Basic Architecture - cont.

< previous

next >

- **PHP 5 Request XML Processing**
  - The PHP 5 CLI script (paxi.psh) reads stdin via output buffering
  - Multipart MIME entities are created and wrapped around each other
    - Unique Boundary values are generated
    - Accurate Content-Length values are determined
  - Managing large amounts of XML quickly and efficiently was a goal; using output buffering provided a fast and flexible method for doing this
- **SOAP Server Communication**
  - Manual SSL socket communication using fsockopen(). Flexibility and performance were key concerns

```
<?php
 $soapfp = fsockopen(SOAPD_URL,SOAPD_PORT,$errno,$errstr,CONNECT_TIMEOUT);
 ?>
```

  - Network and SOAP server health is chaotic and problematic
    - Detection of network/server errors required connection and communication timeouts and retries for both request and response phases
    - PHP 5's stream API stabilized since PHP 4

1 2 3 4 5 6 7 **8** 9 10 11 12 13 toc

The Basic Architecture - cont.

- **PHP 5 Response XML Processing**
  - pMIME accepts a file descriptor (in this case a network socket from the SOAP server) and determines the structure of the incoming MIME/SOAP packet in real-time

```
<?php

$responseparser = new pMIME;
$responseparser->Incoming($soapfp);

?>
```

- pMIME is lightweight and fast, keeping only a single copy of the data. Structure is retained by use of an array of integers
- Particular MIME entities and header fields can be examined. SESSIONID was important for transactional integrity

```
<?php

$responseparser->setHeaderPart(0);
$responseparser->setField('Set-Cookie',TRUE);
if( $responseparser->isParameter('SESSIONID') )
  $REQUEST_SESSIONID = $responseparser->parseField('SESSIONID');
         else
  $REQUEST_SESSIONID = NULL;

?>
```

- Extracted XML is passed to SimpleXML routines for XML parsing and manipulation

```
<?php

$xmlresponse_array = XMLResponseParser($responseparser->fetchPart(5);

?>
```

The Basic Architecture - cont.

- **PHP 5 to PHP 4 Communication**
  - The XML response is often large and the array that is generated is equally large and complex
  - The PHP 4 script expects a string representation of an array. Using serialize() and native UNIX file descriptors make this an efficient operation

```
<?php

in paxi.psh:  echo serialize($xmlresponse_array);

in PHP 4:
    ob_start();
    fpassthru($soappipes[1]);
    $response_array = unserialize(ob_get_clean());

?>
```

  - The presentation logic in PHP 4 now determines formatting and layout of the returned data
  - If data appears invalid or corrupt, the user's original request is resubmitted to paxi.psh from memory and the process starts again

Why SimpleXML

< previous

next >

- To be the first kid on my block using in production
- Lower level to speed up parsing large file size
- Built in XPATH
- It's so easy, even I can do it
- **Note: since this presentation, it is rumoured that many bugs in SimpleXML have been fixed, making many of the workarounds below unnecessary**

```
<?xml version="1.0" encoding="UTF-8"?>
<nyphp>
    <currentVersion>2</currentVersion>
    <userID>NYPHP</userID>
    <memberShip>1256</memberShip>
    <state>New York</state>
    <members>
        <member>
            <memberID>001</memberID>
            <email>hans at nyphp dot org</email>
            <contactInfo>
                <address>
                    <street>123 Street</street>
                    <city>New York</city>
                    <stateID>NY</stateID>
                    <postalCode>10101</postalCode>
                    <countryID>US</countryID>
                </address>
                <phone>212 867 5309</phone>
                <fax/>
            </contactInfo>
        </member>
        <member>
            <memberID>023</memberID>
            <email>harlan at nyphp dot org</email>
            <contactInfo>
                <address>
                    <street>127 Street</street>
                    <city>New York</city>
                    <stateID>NY</stateID>
                    <postalCode>10101</postalCode>
                    <countryID>US</countryID>
                </address>
                <phone>212 666 HELL</phone>
                <fax/>
            </contactInfo>
        </member>
        <member>
            <memberID>066</memberID>
            <email>snyder at nyphp dot org</email>
            <contactInfo>
                <address>
                    <street>185 Street</street>
                    <city>New York</city>
                    <stateID>NY</stateID>
                    <postalCode>10101</postalCode>
                    <countryID>US</countryID>
                </address>
                <phone>212 666 HELL</phone>
                <fax/>
            </contactInfo>
        </member>
    </members>
    <extraStuff>
        <URL>www.nyphp.org</URL>
        <meetingDate>Fourth Tuesday of each Month</meetingDate>
        <comments>Not the last Tuesday</comments>
    </extraStuff>
</nyphp>
```

- Load a string or file into SimpleXML
- Then you can act on the object using SimpleXML methods, looping through the nodes or using XPATH
- In our case we want to rebuild the object into an array so we can normalize the data from the different XML feeds, access it in a variety of ways and place certain values into DB

```
<?php

  /* create SimpleXML object */
  $xml = simplexml_load_string($responsexml);

  /* Find the name of the root node
       Would prefer to do this entirely in SimpleXML */
  $type = dom_import_simplexml($xml)->tagName;

  /* you can also do:

  foreach ($xml as $key=>$value) {
    $type = $key;
  }
     not fully tested */

  /* call the toArray method for this particular XML file (Parser_nyphp class) */
  if ( $type == 'nyphp' ) $response_array = PARSER_nyphp::toArray($xml);


  /* simple parser - Adam Trachtenberg */
    class PARSER_ComplexType {

      protected $data = array();

      static public function toArray() {
        return array();
      }

    }

  /* parser for nyphp node - need to know schema */
    class PARSER_nyphp extends PARSER_ComplexType {

         static public function toArray($xml) {

        $data = array();        /**** protected $data ****/

      /* Need to test if a node exists.
         Two possible solutions:

         a) not tested - Adam? */
      if ( count($xml->xpath(currentVersion)) > 0 ) {
        $data['currentVersion'] = (int) $xml->currentVersion;
      }

      /* b) we can only do this on a leaf node, will the above always work -
 what about with iterators (as below)???  */
      if ( (string) $xml->currentVersion) !='' ) {
        $data['currentVersion'] = (int) $xml->currentVersion;
      }
```

```
        if ( (string) $xml->userID) !='' ) {
          $data['userID'] = (string) $xml->userID;
        }

        .
        .
        .

        /* Must be a better way to do this???

           Right now if you cast a node that has children to a string
           it returns as an empty string, thus you need to test for the
           leaf node, which will return the value
        */
        if ( (string) $xml->members->member->memberID !='' ) {

              foreach($xml->members as $member) {
                $data['members'][] = PARSER_member::toArray($member);
              }

            }

        .
        .
        .

           return $data;

          }

      }

      /* build out a class for each node */
      class PARSER_member extends PARSER_ComplexType {

           static public function toArray($xml) {

          $data = array();

        if ( (string) $xml->memberID) !='' ) {
          $data['memberID'] = (int) $xml->memberID;
        }

        if ( (string) $xml->email) !='' ) {
          $data['email'] = (string) $xml->email;
        }

        .
        .
        .

        /* same as above, need to test the leaf */
        if ( (string) $xml->contactInfo->address->street !='') {

              foreach($xml->contactInfo->address as $address) {
                // here we alter the way the array is returned, leaving out the contactInfo node
                $data['address'] = PARSER_address::toArray($address);
              }

            }

        .
        .
        .

           return $data;

          }

      }

     /* build out a class for each node */
      class PARSER_address extends PARSER_ComplexType {

           static public function toArray($xml) {

          $data = array();
        .
        .
        .
        return $data;
        }
      }
?>
```

- Some limitations exist, and some functionality needs to be added to SimpleXML
- But if you know the Schema, it's fast and easy to build out classes to build any structure you need to work with
- You can also easily work directly with the SimpleXML object

The Next Generation of PAXI

- We use PHP sessions to maintain state, and a response table to tie remote responses to sessions.
    - The remote request script is called with a key that it will use to save the remote response
    - Waiting.php script looks for the returned response, refreshing every few seconds
    - If the response times out, the waiting script redisplays the current step in the process, otherwise it uses the information in the response to display the next step.
- Remote requests may now be called in advanced, and saved for later use by the session

References

- **PHP / Javascript:** www.webxpertz.net/faqs/jsfaq/jsserver.php
- **SimpleXML:** www.php.net/simplexml
- **SOAP:** www.w3.org/TR/2003/REC-soap12-part1-20030624

Presentation given by: Christopher Hendry (chendry at harlangroup dot org)

Table of Contents

< previous

1 2 3 4 5 6 7 8 9 10 11 12 13 **toc**