By icarus

Table of Contents

Speaking In Tongues

As a Web developer, you probably already know how frustrating it is to get applications written in different languages to talk nice to each other. Each language has its own data types, its own syntax, and its own creative twist on standard programming constructs, and it's no small task to get them to cooperate with each other. In fact, in the multi−platform, multi−lingual world of the Web, exchanging data between applications written in different programming languages is often the biggest stumbling block to building new, integrated Web services.

Well, fear not − there does exist a way. It comes in the form of Web Distributed Data eXchange (WDDX), an XML−based technology designed to create language− and platform−independent representations of data, with a view to simplifying data exchange over the Web.

Over the next few pages, I'm going to give you a quick primer on how WDDX works, and then combine it with my favourite Web programming language, Perl, in order to demonstrate how you can use it to build simple applications. So keep reading − things are just getting interesting!

The Big Picture

It's quite likely that you've never heard of WDDX before, so let me take a couple minutes to give you a quick overview.

WDXX, or Web Distributed Data eXchange, is a mechanism for representing and exchanging data structures (like strings and arrays) between different platforms and applications. It uses XML to create abstract representations of data, application−level components to convert the XML abstractions into native data structures, and standard Internet protocols like HTTP and FTP as the transport mechanism between applications.

Still confused, huh?

All right, let's try English for a change.

WDDX is simply a way of representing data − strings, numbers, arrays, arrays of arrays − in a manner that can be easily read and understood by any application. To illustrate this, consider the following PHP variable:

```
$str = "Robin Hood";
```

Here's how WDDX would represent this variable:

```
<wddxPacket version='1.0'>
<header/>
<data>
<string>Robin Hood</string>
</data>
</wddxPacket>
```

By creating an abstract representation of data, WDDX makes it possible to easily exchange data between different applications − even applications running on different platforms or written in different languages − so long as they all understand how to decode WDDX−encoded data. Applications which support WDDX will receive application−neutral WDDX data structures and convert them into an application−specific usable format. In other words, a Perl hash could be converted to a WDDX representation and sent to a Python script, which would decode it into a native Python list, or a PHP array could be sent to a JSP script, where it would be correctly recognized and handled.

By maintaining the integrity of data structures across different environments, writing platform−independent code becomes much easier. And, since WDDX data representations are nothing but regular XML, they can be transferred over any protocol that supports ASCII text − HTTP, email, FTP and so on. This makes WDDX both platform−neutral and portable − two attributes that have immediately endeared it to the developer community.

Packet Sniffer

All WDDX "packets" are constructed in a standard format.

The root, or document, element for WDDX data is always the <wddxPacket> element, which marks the beginning and end of a WDDX block.

```
<wddxPacket version='1.0'>
```

This is immediately followed by a header containing comments,

```
<header>
<comment>This packet was generated on stardate 56937,
constellation
Omega </comment> </header>
```

and a data area containing WDDX data structures.

```
<data>
...
</data>
</wddxPacket>
```

In order to perform its magic, WDDX defines a set of base data types that correspond to the data types available in most programming languages. Here's a list, with examples − pay attention, because you'll be seeing a lot of these in the next few pages:

Strings, represented by the element <string> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<string>Robin Hood</string>
</data>
</wddxPacket>
```

Numbers, represented by the element <number> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<number>5</number>
</data>
</wddxPacket>
```

Boolean values, represented by the element <boolean> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<boolean value='true'/>
</data>
</wddxPacket>
```

Timestamps, represented by the element <dateTime> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<dateTime>2002−06−08T16:48:23</dateTime>
</data>
</wddxPacket
```

Arrays and hashes, represented by the elements <array> and <struct> respectively − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<array length='3'>
<string>red</string>
<string>blue</string>
<string>green</string>
</array>
</data>
</wddxPacket>
```

Tabular data, represented by the element <recordset> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<recordset rowCount='3' fieldNames='ID,NAME'>
<field name='ID'>
<number>1</number>
<number>2</number>
<number>3</number>
</field>
<field name='NAME'>
<string>John</string>
<string>Joe</string>
<string>Mary</string>
</field>
</recordset>
</data>
</wddxPacket>
```

Base64−encoded binary data, represented by the element <binary> − for example

```
<wddxPacket version='1.0'>
<header/>
<data>
<binary length='24'>VGhpcyBpcyBzb21lIGJpbmFyeSBkYXRh</binary>
</data>
</wddxPacket>
```

The process of converting a data structure into WDDX is referred to as "serialization". The process of decoding a WDDX packet into a usable form is, obviously, "deserialization". The serializer/deserializer component is usually built into the programming language − as you will see on the next page, when I introduce Perl into the equation.

Boyz 'N The Hood

Perl implements WDDX via its WDDX.pm module, which you can download from http://www.scripted.com/wddx/. This Perl module includes both a serializer and deserializer for WDDX, and also implements all the base data types defined in the WDDX specification.

The basic procedure for serializing a WDDX packet with WDDX.pm is fairly straightforward: first create an object representation of the value you want to serialize, and then call the module's serialize() method to actually create the packet data. The following example illustrates the process of serializing a string variable with the module's string() method:

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
my $wddx = new WDDX;

# create WDDX string object
$obj = $wddx->string("Robin Hood");

# serialize and print object
print $wddx->serialize($obj);
```

Here's the output:

```
<wddxPacket version='1.0'>
<header/>
<data>
<string>Robin Hood</string>
</data>
</wddxPacket>
```

In a similar manner, you can use the number() and boolean() methods to create numeric and Boolean data representations also.

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
```

```
my $wddx = new WDDX;

# create WDDX number object
$obj = $wddx->number(5);

# serialize and print object
print $wddx->serialize($obj);
```

You can serialize integer-indexed arrays with the array() method.

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
my $wddx = new WDDX;

# create a Perl array containing WDDX data objects
my @colors = (
$wddx->string("red"),
$wddx->string("blue"),
$wddx->string("green"),
);

# create a WDDX array object
# note that the array() method must be passed a reference
$obj = $wddx->array(\@colors);

# serialize and print object
print $wddx->serialize($obj);
```

Note that, when serializing arrays, the array() method must be passed a reference to a Perl array containing WDDX data objects. These objects need not be of the same type.

You can also serialize a Perl hash with the struct() method.

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
my $wddx = new WDDX;
```

```
# create WDDX struct from a Perl hash
# note that values of the Per,l hash must be WDDX data objects
$obj =
$wddx->struct(
{
"parrot" => $wddx->string("Polly"),
"hippo" => $wddx->string("Harry"),
"iguana" => $wddx->string("Ian")
}
);

# serialize and print object
print $wddx->serialize($obj);
```

This creates a <struct> containing name−value pairs corresponding to the elements of the hash.

```
<wddxPacket version='1.0'>
<header/>
<data>
<struct>
<var name='hippo'>
<string>Harry</string>
</var>
<var name='parrot'>
<string>Polly</string>
</var>
<var name='iguana'>
<string>Ian</string>
</var>
</struct>
</data>
</wddxPacket>
```

Finally, you can generate a WDDX <recordset> with the recordset() method. This method accepts three arguments: a list of column names, a list of corresponding data types, and a two−dimensional list of values. Take a look at the following example, which might make this clearer:

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
my $wddx = new WDDX;
```

```
# create WDDX recordset
$obj = $wddx->recordset(
["ID", "NAME"],
["number", "string"],
[
[1, "John"],
[2, "Joe"],
[3, "Mary"]
]
);

# serialize and print object
print $wddx->serialize($obj);
```

You can create binary data packets with the binary() method, and timestamps with the datetime() method − I'll leave these to you to experiment with.

All Mixed Up

The flip side of whatever you just read is, of course, deserialization. Perl's WDDX.pm module accomplishes this via its deserialize() method, which can be used to convert WDDX−based language−independent data representations into native data types.

Consider the following example, which demonstrates how a WDDX packet containing a string value is deserialized into a Perl scalar variable:

```
#!/usr/bin/perl

# use WDDX module
use WDDX;

# create WDDX object
$wddx = new WDDX;

# simulate a WDDX packet
$packet = "<wddxPacket
version='1.0'><header/><data><string>Robin
Hood</string></data></wddxPacket>";

# deserialize packet into WDDX string object
$obj = $wddx->deserialize($packet);

# check object type
if ($obj->type eq "string")
{
# and print as scalar
print $obj->as_scalar;
}
```

In this case, the WDDX packet is first deserialized into a WDDX string object, and then the string object's as_scalar() method is used to convert the string object into a native Perl scalar. Note that the deserialized object exposes a type() method, which can be used to identify the data type and process it appropriately.

Here's the output:

```
Robin Hood
```

This deserialization works with arrays too − as the following example demonstrates:

```
#!/usr/bin/perl

# use WDDX module
use WDDX;

# create WDDX object
$wddx = new WDDX;

# simulate a WDDX packet
$packet = "<wddxPacket version='1.0'> <header/> <data> <array
length='3'> <string>red</string> <string>blue</string>
<string>green</string> </array> </data> </wddxPacket>";

# deserialize packet into WDDX array object
$obj = $wddx->deserialize($packet);

# get number of elements in array
$length = $obj->length();

# get reference to native Perl array
$arrayref = $obj->as_arrayref();

# iterate through array and print elements
for ($i=0; $i<$length; $i++)
{
print "$$arrayref[$i]\n";
}
```

Here's the output:

```
red
blue
green
```

Wanna really cause some heartburn? Try serializing an array of arrays,

```
#!/usr/bin/perl

# include module
use WDDX;

# create WDDX object
my $wddx = new WDDX;

# create an array
```

```
my @arr = ( $wddx->string("huey"), $wddx->string("dewey"),
$wddx->boolean(1) );

# create a WDDX hash
$obj = $wddx->struct(
{
"str" => $wddx->string("Abracadabra"),
"num" => $wddx->number(23),
"arr" => $wddx->array(\@arr)
}
);

# serialize and print object
print $wddx->serialize($obj);
```

and see what you get:

```
<wddxPacket version='1.0'>
<header/>
<data>
<struct>
<var name='num'>
<number>23</number>
</var>
<var name='str'>
<string>Abracadabra</string>
</var>
<var name='arr'>
<array length='3'>
<string>huey</string>
<string>dewey</string>
<boolean value='true'/>
</array>
</var>
</struct>
</data>
</wddxPacket>
```

This is a hash with three keys, one containing a string, the second a number, and the third an array. Now, try deserializing the WDDX packet generated from the code above.

```
#!/usr/bin/perl

# use WDDX module
use WDDX;
```

```
# create WDDX object
$wddx = new WDDX;

# simulate a WDDX packet
$packet = " <wddxPacket
version='1.0'><header/><data><struct><var
name='num'><number>23</number></var><var
name='str'><string>Abracadabra</string></var><var
name='arr'><array
length='3'><string>huey</string><string>dewey</string><boolean
value='true'/></array></var></struct></data></wddxPacket>";

# deserialize packet into WDDX array object
$obj = $wddx->deserialize($packet);

# get reference to native Perl hash
$hashref = $obj->as_hashref();

# get keys
@k = $obj->keys();

# print keys and type of corresponding values
foreach $k (@k)
{
print "$k --> " . $obj->get($k)->type . "\n";
}
```

Here's what you should see:

```
num --> number
str --> string
arr --> array
```

Flying Toasters And Dancing Knives

Now that you've understood the fundamentals, let's look at a simple application of WDDX.

One of the most popular uses of WDDX involves using it to retrieve frequently−updated content from a content repository − news headlines, stock prices and the like. Since WDDX is designed expressly for transferring data in a standard format, it excels at this type of task − in fact, the entire process can be accomplished via two very simple scripts, one running on the server and the other running on the client.

Let's look at the server component first. We'll begin with the assumption that the content to be distributed (products and their corresponding prices, in this case) are stored in a single database table, which is updated on a regular basis. The table in which this data is stored looks a lot like this:

```
+----+---------------------------+--------+
| id | name | price |
+----+---------------------------+--------+
| 1 | XYZ Slicer−Dicer−Toaster | 123.78 |
| 2 | FGH Serrated Carving Knife | 34.99 |
| 3 | Dual−Tub Washing Machine | 455.99 |
+----+---------------------------+--------+
```

Now, we need to write a script which will reside on the server, connect to this table, retrieve a list of items in the catalog, and encode them as a WDDX packet

```
#!/usr/bin/perl

# load modules
use DBI;
use WDDX;

# create WDDX object
$wddx = new WDDX;

# connect
my $dbh =
DBI−>connect("DBI:mysql:database=mydb;host=localhost", "root",
"gd63hrd", {'RaiseError' => 1});

# execute query
my $sth = $dbh−>prepare("SELECT * FROM catalog");
$sth−>execute();

# counter
my $count = 0;
my @data;
```

```
# iterate through resultset
# append record to @data array as WDDX data objects
# each element of this array is a hash with
# keys corresponding to the column names
while(my $ref = $sth->fetchrow_hashref())
{
$data[$count] = $wddx->struct({ "id" =>
$wddx->number($ref->{'id'}), "name" =>
$wddx->string($ref->{'name'}),
"price" => $wddx->number($ref->{'price'}) });
$count++;
}

# clean up
$dbh->disconnect();

# create a WDDX array data object from @data
$obj = $wddx->array(\@data);

print "Content-type: text/html\r\n\r\n";

# serialize and print the packat
print $wddx->serialize($obj);
```

This is a pretty simple script, especially if you know a little bit about Perl. First, I've included both the WDDX module and the additional DBI module (which I need in order to retrieve data from the database). I've then used DBI's standard constructs to connect to the database and retrieve all the records from the "catalog" table.

Once a resultset has been retrieved, a "while" loop is used to iterate through it. Each record is retrieved as a Perl hash, and the various elements (columns) of each record are then restructured into a Perl hash containing three keys − "id", "name" and "price" − and WDDX−encoded values. This hash is then converted into a WDDX <struct> via the struct() method, and added to the @data Perl array.

Once all the records in the resultset have been processed, the @data Perl array (which now contains all the data from the table as a series of WDDX <struct>s) is converted into a WDDX array via the array() method, serialized via the serialize() method, and printed to the standard output.

Here's a sample WDDX packet generated by the server above:

```
<wddxPacket version='1.0'><header/><data><array
length='3'><struct><var
name='name'><string>XYZ
Slicer-Dicer-Toaster</string></var><var
name='price'><number>123.78</number></var><var
name='id'><number>1</number></var></struct><struct><var
name='name'><string>FGH Serrated Carving
```

```
Knife</string></var><var
name='price'><number>34.99</number></var><var
name='id'><number>2</number></var></struct><struct><var
name='name'><string>Dual-Tub Washing
Machine</string></var><var
name='price'><number>455.99</number></var><var
name='id'><number>3</number></var></struct></array></data></wddxPacket>
```

This script should be placed in your Web server's "cgi-bin" directory and given appropriate permissions, so that a user accessing it via a Web browser sees a WDDX packet like the one above. Every time a user accesses the script, a new WDDX packet, containing the latest data from the database, is generated and printed.

Different Strokes

So that takes care of the server − now how about the client?

The function of the client should be fairly obvious − connect to the server described above via HTTP, read the WDDX packet generated, deserialize it into a native representation, and display the output to the user. Here's the script to accomplish this:

```
#!/usr/bin/perl

# module to read HTTP response
use LWP::UserAgent;

# WDDX module
use WDDX;

# create an HTTP client
$client = LWP::UserAgent->new;
my $request = HTTP::Request->new(GET =>
"http://my.wddx.server/cgi-bin/server.cgi");

# store the response in an object
my $result = $client->request($request);

# get a good response
if ($result->is_success)
{
# deserialize resulting packet as array and get reference
my $wddx = new WDDX;

$packet = $wddx->deserialize($result->content);
$arrayref = $packet->as_arrayref;
$length = $packet->length();

# iterate through array
for($x=0; $x<$length; $x++)
{
# get hash reference and print values
$item = $$arrayref[$x];
print "ID: " . $item->{'id'} . "\nName: " .
$item->{'name'} . "\nPrice: " . $item->{'price'} . "\n\n";
}

}
# response is bad...
else
{
```

```
print "Could not read response";
}
```

Here, I've used the LWP Perl module to instantiate a simple HTTP client, which I've then pointed at the URL of the WDDX server. This client connects to the specified URL, retrieves the dynamically−generated WDDX packet, and stores it in an object.

This object is then passed to the WDDX module's deserialize() method, and is converted into its native form, a Perl array. A "for" loop is used to iterate through the array, access each individual hash within the array, and retrieve the data for each item in the catalog. This data is then printed to the standard output device.

And, when you run this script, you'll see something like this:

```
ID: 1
Name: XYZ Slicer−Dicer−Toaster
Price: 123.78

ID: 2
Name: FGH Serrated Carving Knife
Price: 34.99

ID: 3
Name: Dual−Tub Washing Machine
Price: 455.99
```

It's important to note that this client need not be on the same physical machine as the server − so long as it can connect to the server via HTTP, it can retrieve the WDDX packet, decode it and display the results. Heck, it doesn't even need to be in Perl − here's another version of the same client, this time in PHP.

```
<?
// url of Web page
$url = "http://my.wddx.server/cgi−bin/server.cgi";

// read WDDX packet into string
$packet = join ('', file($url));

// deserialize
$data = wddx_deserialize($packet);
?>
<html>
<head>
<basefont face="Verdana">
</head>
<body>
```

```
<table border="1" cellspacing="0" cellpadding="5">

<tr>
<td>Item ID</td>
<td>Name</td>
<td>Price</td>
</tr>

<?
// iterate through array
foreach($data as $item)
{
?>
<tr>
<td><?=$item['id']?></td>
<td><?=$item['name']?></td>
<td><?=$item['price']?></td>
</tr>
<?
}
?>
</table>

</body>
</html>
```

Here's what it looks like

Item ID	Name	Price
1	XYZ Slicer-Dicer-Toaster	123.78
2	FGH Serrated Carving Knife	34.99
3	Dual-Tub Washing Machine	455.99

Now that's what I call platform−independence!

This Way Out

And that's about it for the moment. In this article, I gave you a quick tour of WDDX, an innovative, XML−based technology designed to simplify data exchange over the Web. I showed you how you could use Perl, with its supplementary WDDX component, to serialize and deserialize a variety of different data structures, including scalars, arrays and hashes. Finally, I wrapped thngs up with a real−life demonstration of the power of WDDX, building a client−server application to transmit and retrieve WDDX−encoded information...using two different programming languages.

Needless to say, you've had a busy afternoon. Take some time off, get yourself a cup of coffee, and when you get back, drop by the following links, which have more information on WDDX.

The WDDX Web site, at http://www.openwddx.org/

The WDDX DTD, at http://www.allaire.com/documents/objects/whitepapers/wddx_dtd.txt

The WDDX.pm Perl module, at http://www.scripted.com/wddx/

PHP's WDDX functions, at http://www.php.net/manual/en/ref.wddx.php

Moreover.com, which offers free news headlines in WDDX format (non−commercial use only), at http://www.moreover.com/

Until next time...stay healthy!

Note: All examples in this article have been tested on Linux/i586 with Perl 5. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!