NAME

psptut - a tutorial on writing PSP websites

DESCRIPTION

A introducing tutorial on writing PSP enabled websites using PSP's CGI frontend engine psp.cgi.

PSP Code Blocks and Data Output

Direct Output and Embedded Perl Blocks

PSP code usually consists of one or more parts of these two classes:

1. Direct output; i.e. (X)HTML code that is directly passed trough the engine into PSP output.

2. Embedded Perl code; Perl code as usual but with some extra objects and data structures exported by PSP::Engine.

By default, PSP::Engine handles everything that is not embedded Perl code as direct output, embedded Perl code is written in code blocks. These code blocks are initiated by strings that are configurable via the CGI::Config object class. By name, these are OpenTag ("<%" by default) and CloseTag ("%>" by default).

A PSP::Engine object also exports itself into the executed code, so you may access all the methods described above.

This PSP code example generates a simple HTML page and prints a "hello world":

```
<html>
 <body>
  <% $psp->print("hello world!"); %>
 </body>
</html>
```

Executing this code results in:

```
<html>
 <body>
  hello world!
 </body>
</html>
```

This method of generating output has 2 downsides:

1. It is quite long winded.

2. You have to take care to not create applications that are vulnerable to XSS (Cross Site Scripting) attacks.

The latter is very imporant and most other embedded code languages/interpreters, (at the head PHP) have become infamous for being susceptible to such attacks.

Preventing Cross Site Scripting

Imagine the following PSP code and a situation, where $a is read from i.e. the URL or from a database where people store their email addresses or a file or something else other people may have write access to:

```
<html>
 <body>
  <% $psp->print($a); %>
 </body>
```

```
</html>
```

Now $a is read from a database where someone has stored "click here" instead of an email address.

Now executing the code above, it will result in this HTML code:

```
<html>
 <body>
  <a href="attackerswebsite">click here</a>
 </body>
</html>
```

This does not look very critical at a first sight, but it could also be a login form misguiding people to send their user/password combinations to the attacker or a JavaScript that modifies an existing login form to send data to the attacker.

To prevent such problems, the *PSP::CGI::Transform* package exports among other functions also the *htmlize function* that converts any text string to HTML/XHTML compliant and safe data.

So by modifying the PSP code above to use htmlize() we eliminate the problem:

```
<%
 use PSP::CGI::Transform;
 my $a='<a href="attackerswebsite">click here</a>';
%>
<html>
 <body>
  <% $psp->print(htmlize($a)); %>
 </body>
</html>
```

Executing this code now results in the following output that is completely HTML/XHTML XSS safe:

```
<html>
 <body>
  &lt;a href=&quot;attackerswebsite&quot;&gt;click here&lt;/a&gt;
 </body>
</html>
```

Data Output Shortcuts

Now as the XSS problem is solved, there is still our first problem; "printing" data to the PSP output this way is long winded.

To solve this, variations of the OpenTag ("<%") were created as shortcuts: The PrintSecureOpenTag ("<%=" by default) and the PrintOpenTag ("<%==" by default).

If you want to print data that is possibly untrusted, using one of these two lines is allowed:

```
<%= $thisMayContainInsecureText %>

<% $psp->print(htmlize($thisMayContainInsecureText)); %>
```

Both lines do the same; sanitizing the text using htmlize() and printing the results to the PSP output queue.

Likewise these two PSP code lines are the same and may only be used for data that is really trusted and/or if you want to print trusted HTML text:

```
<%== $thisContainsSecureText %>

<% $psp->print($thisContainsSecureText); %>
```

You are strongly advised to use the PrintSecureOpenTag ("<%=") whenever it is possible for data output to prevent XSS pitfalls.

Block-Overlapping Code

The way PSP::Engine handles direct output code allows, unlike most other embedded programming solutions, code that spans over several blocks.

For example, code like this is valid and used very often:

```
<html>
 <body>
<%
 if($a > 0){
%>
  <h1>$a is greater than 0!</h1>
<%
 }
 else{
%>
  <h1>$a is less or equal 0!</h1>
<%
 }
%>
 </body>
</html>
```

For $a > 0, this results in

```
<html>
 <body>

  <h1>$a is greater than 0!</h1>

 </body>
</html>
```

and for $a <= 0 in

```
<html>
 <body>

  <h1>$a is less or equal 0!</h1>

 </body>
</html>
```

You may even use loops with for, foreach, while, until, etc.:

```
<%
```

```
 my $data = [
   {
    first_name => 'John R.',
    last_name  => 'Doe',
    age        => 39
   },
   {
    first_name => 'Jane F.',
    last_name  => 'Doe',
    age        => 22
   },
   {
    first_name => 'John Q.',
    last_name  => 'Public',
    age        => 28
   }
  ];
%>
<html>
 <body>
  <table>
   <tr>
    <th>Last Name</th>
    <th>First Name</th>
    <th>Age</th>
   </tr>
<%
 foreach my $person (@{$data}){
%>
   <tr>
    <td><%= $person->{last_name} %></td>
    <td><%= $person->{first_name} %></td>
    <td><%= $person->{age} %></td>
   </tr>
<%
 }
%>
  </table>
 </body>
</html>
```

The example above executes into the following output:

```
<html>
 <body>
  <table>
   <tr>
    <th>Last Name</th>
    <th>First Name</th>
    <th>Age</th>
   </tr>

   <tr>
    <td>Doe</td>
    <td>John R.</td>
    <td>39</td>
   </tr>
```

```
    <tr>
     <td>Doe</td>
     <td>Jane F.</td>
     <td>22</td>
    </tr>

    <tr>
     <td>Public</td>
     <td>John Q.</td>
     <td>28</td>
    </tr>

   </table>
  </body>
 </html>
```

Including PSP Files and Data Interchange

Using the file Method for File Inclusion

Recurring, reusable and/or topically independent code parts are predestined for abstraction and inclusion.

As an example, we use a header and a footer file:

header.psp:

```
<html>
 <head>
  <topic>header/footer inclusion example</topic>
 </head>
 <body>
  <h1>Content</h1>
  <ul>
   <li><a href="content1.psp">Content</a></li>
   <li><a href="content2.psp">Other Content</a></li>
  </ul>
```

footer.psp:

```
  </body>
 </html>
```

content1.psp:

```
<% $psp->file('header.psp'); %>
  <h1>This is Content</h1>
  <p>
   Blah blah blah.
  </p>
<% $psp->file('footer.psp'); %>
```

content2.psp:

```
<% $psp->file('header.psp'); %>
  <h1>This is Other Content</h1>
  <p>
   Foo bar foobar.
```

```
  </p>
<% $psp->file('footer.psp'); %>
```

Executing content1.psp now results in:

```
<html>
 <head>
  <topic>header/footer inclusion example</topic>
 </head>
 <body>
  <h1>Content</h1>
  <ul>
   <li><a href="content1.psp">Content</a></li>
   <li><a href="content2.psp">Other Content</a></li>
  </ul>
  <h1>This is Content</h1>
  <p>
   Blah blah blah.
  </p>
 </body>
</html>
```

While content2.psp evaluates to:

```
<html>
 <head>
  <topic>header/footer inclusion example</topic>
 </head>
 <body>
  <h1>Content</h1>
  <ul>
   <li><a href="content1.psp">Content</a></li>
   <li><a href="content2.psp">Other Content</a></li>
  </ul>
  <h1>This is Other Content</h1>
  <p>
   Foo bar foobar.
  </p>
 </body>
</html>
```

It is important to know that executing an include from another directory changes the current working directory to the one from where the file is loaded, so when that file includes other files, loads modules using `use` or `require` or uses filesystem operations such as `open()`, they will apply to the directory the current PSP file is in.

You also have to know that including files with a leading slash ("/") are NOT absolute to the filesystem but relative to the BaseDirectory configuration option. This is usually the webserver's/vhost's DocumentRoot or the psp-root. If you want to use absolute paths, set the BaseDirectory to ''.

See *"file Method" in PSP::Engine* for details.

Handing Over Data to Includes

The *file method* provides the possibility to hand over an argument variable "$args" to the file being included.

For saving stack memory and as only one argument is passed to the included file, it is

intended and recommended that you use an array or hash reference or an anonymous array/hash:

Both array methods

```
# passing an existing array as reference
my @data = ('a', 'b');
$psp->file('file.psp', \@data);
```

or

```
# using an anonymous array for hand-over
$psp->file('file.psp', ['a', 'b']);
```

may be accessed from within file.psp through:

```
my $foo = $args->[0];
my $bar = $args->[1];
```

The data passed by both hash methods

```
# passing an existing hash as reference
my %data = (foo => 'a', bar => 'b');
$psp->file('file.psp', \%data);
```

and

```
# using an anonymous hash for hand-over
$psp->file('file.psp', {foo => 'a', bar => 'b'});
```

can be accessed through:

```
my $foo = $args->{foo};
my $bar = $args->{bar};
```

Here is a modified version of the example above that hands over a document title and a background color for the body to the header.psp.

header.psp:

```
<html>
 <head>
  <topic><%= $args->{title} %></topic>
 </head>
 <body bgcolor="<%= $args->{bgcolor} %>">
  <h1>Content</h1>
  <ul>
   <li><a href="content1.psp">Content</a></li>
   <li><a href="content2.psp">Other Content</a></li>
  </ul>
```

footer.psp:

```
  </body>
</html>
```

content1.psp:

```
<%
 $psp->file('header.psp', {
   title   => 'Welcome to content1.psp',
   bgcolor => '#ff00ff'
  });
%>
  <h1>This is Content</h1>
  <p>
   Blah blah blah.
  </p>
<% $psp->file('footer.psp'); %>
```

See *"file Method" in PSP::Engine* for details.

Storing Data Inside the $psp Object

A PSP::Engine object provides the methods *var* and *setvar* that allow accessing and storing data inside the $psp object. Also the "$var" variable is provided for direct access to the storage hash.

Data stored inside the $psp object is preserved amongst all PSP code being executed by this object. This is especially useful for data required by many different includes, for example a configuration hash.

In this example we use an include "database.psp" to establish a database connection and place the DBI object inside the $psp variables storage from where it is used by other includes:

database.psp:

```
<%
 use DBI;
 my $dbh = DBI->connect(...);
 if(defined($dbh)){
  # connection established
  $psp->setvar('db', $dbh);
 }
 else{
  # connection failed
  $psp->print('Error connecting to the database!');
 }
%>
```

content.psp:

```
<%
 # connect to database first
 $psp->file('database.psp');
 my $dbh = $psp->var('db');
%>
<html>
 <body>
<%
 if(defined($dbh)){
  my $sth = $dbh->prepare(
    'SELECT first_name, last_name, comments FROM persons'
   );
  $sth->execute();
  while($sth->fetchrow_hashref()){
%>
```

```
  <h1><%= $_->{first_name}.' '.$_->{last_name} %></h1>
  <p>
   <%= $_->{comments} %>
  </p>
<%
  }
 }
%>
 </body>
</html>
```

For storing data inside the $psp object, the following statements are equivalent:

```
$psp->setvar('foo','bar');

$psp->var->{foo} = 'bar';

$var->{foo} = 'bar';
```

Likewise you have several variants for accessing stored data. All these statements do exactly the same:

```
my $value = $psp->var('foo');

my $value = $psp->var->{foo};

my $value = $var->{foo};
```

For more information on storing variables inside PSP::Engine objects, see *PSP::Engine*.

Using CGI Extensions

For PSP, CGI operations are supplied by an object of the the *PSP::CGI* class, accessable through `$psp->cgi` and (for convenience) also through `$cgi`.

Accessing CGI Parameters and Form Data

An essential element for writing web applications is access to parameters and data supplied by the web browser.

The following XHTML code will show a simple form and transfer the data to the PSP script "receive.psp":

```
<html>
 <body>
  <form action="receive.psp" method="GET">
   <p>
    Please enter your first name:<br />
    <input type="text" name="firstname" /><br />
    And your last name:<br />
    <input type="text" name="lastname" /><br />
    <input type="submit" value="Send!" />
   </p>
  </form>
 </body>
</html>
```

Pay attention to the `method="GET"` attribute in the `<form>` tag. This means, the data is to be URI encoded. Entering "John Q." for the first, "Public" for the last name and pressing the "Send!" button will cause the web browser to browse to "receive.psp?firstname=John+Q.&lastname=Public".

It is obvious this is neither sensible use for bigger amounts of data, as the maximum length of an URL is restricted, nor for sensitive data, as URLs are stored in the web browser's history and logged on proxy and web servers.

So for bigger and/or more sensitive data (personal data such as real names should already be respected as sensitive data), you should use the HTTP POST method:

```
<html>
 <body>
  <form action="receive.psp" method="POST">
   <p>
    Please enter your first name:<br />
    <input type="text" name="firstname" /><br />
    And your last name:<br />
    <input type="text" name="lastname" /><br />
    <input type="submit" value="Send!" />
   </p>
  </form>
 </body>
</html>
```

Regardless of the method you used to transfer the HTTP transmit method you can now access this data through the *$cgi->param() method*, as this example shows:

```
<%
 my $firstname=$cgi->param('firstname');
 my $lastname=$cgi->param('lastname');
%>
<html>
 <body>
<%
 if(defined($firstname) && defined($lastname)){
%>
  <p>
   Welcome <%= $firstname.' '.$lastname %>!
  </p>
<%
 }
 else{
%>
  <p>
   Go back and fill in the complete form!
  </p>
<%
 }
%>
 </body>
</html>
```

Instead of

```
 my $firstname=$cgi->param('firstname');
 my $lastname=$cgi->param('lastname');
```

you could also use direct access to the CGI parameter hash,

```
my $firstname=$cgi->param->{firstname};
my $lastname=$cgi->param->{lastname};
```

whatever is most suitable for your code or does your usual coding style match best. Also remember you may use `$psp->cgi->param` instead of `$cgi->param`.

The following example shows the possibility of using the *$cgi->params() method* for creating a list of all CGI parameters and their values supplied:

```
<html>
 <body>
  <h1>CGI Parameters</h1>
  <table>
   <tr>
    <th>Name</th>
    <th>Value</th>
   </tr>
<%
 foreach(sort($cgi->params)){
%>
   <tr>
    <td><%= $_ %></td>
    <td><%= $cgi->param($_) %></td>
   </tr>
<%
 }
%>
  </table>
 </body>
</html>
```

Multiple Parameters of the Same Name

In many situations it is required to call a script with multiple parameters of the same name, i.e. (X)HTML forms that allow the selection of more than one element of the same name.

If PSP::CGI receives a parameter of the same name twice or more, the values are appended to the predecessors, separated by a ASCII NUL character (0x00). To release you from dealing with this, PSP::CGI provides the *multiparam method* that returns an array with one element per value instead of a single string.

Have a look at this XHTML example:

```
<html>
 <body>
  <form action="multiselect.psp" method="POST">
   <h1>Select the extra ingredients for your pizza</h1>
   <p>
    <select name="extras" size="3" multiple="multiple">
     <option value="cheese">Extra Cheese</option>
     <option value="pepperoni">Pepperoni</option>
     <option value="tuna">Tuna</option>
     <option value="anchovies">Anchovies</option>
     <option value="garlic">Garlic</option>
     <option value="peppers">Sweet Peppers</option>
     <option value="hotpeppers">Hot Peppers</option>
    </select><br />
```

```
    <input type="submit" value="Order!" />
   </p>
  </form>
 </body>
</html>
```

For cases like the multiple options form above, `$cgi->multiparam()` allows easy evaluation multiple parameters of the same name, as this example shows:

```
<html>
 <body>
  <h1>Extras chosen</h1>
  <ul>
<%
 foreach($cgi->multiparam('extras')){
%>
   <li><%= $_ %></li>
<%
 }
%>
  </ul>
 </body>
</html>
```

Getting and Setting Cookies

Cookies are short text strings stored on the clients side, usually in a file called cookies.txt, that are transmitted back to the server when a corresponding website is visited. The server may use them to store recognition data on the client side, for example user tracking IDs, authentication credentials or status information.

Look at this example, showing you how to list, request and store cookies:

```
<html>
 <body>
  <h1>Cookies</h1>
  <table>
   <tr>
    <th>Name</th>
    <th>Value</th>
   </tr>
<%
 foreach(sort($cgi->cookies)){
%>
   <tr>
    <td><%= $_ %></td>
    <td><%= $cgi->cookie($_) %>
   </tr>
<%
 }
 $cgi->setcookie('test1','foo',5);
 $cgi->setcookie('test2','bar',10);
%>
  </table>
  <p>
   Two cookies were set now: test1=foo (5 seconds), test2=bar (10
seconds)
  </p>
```

```
   <p>
    Reload page to check.
   </p>
  </body>
 </html>
```

The *cookies method* gets a list of the names of all cookies the web browser sent (back) to the server. This is much like the *params method* we discussed before.

The same applies to the *cookie method* which resembles the *param method* but returns the value of a given cookie name instead of a CGI parameter. Equally you may use the `$cgi->cookie->{$key}` syntax to gain direct access to the PSP::CGI object's cookie hash.

Setting a cookie is somewhat more complex, although the example above impressively shows how easy setting a cookie can be using PSP. The 3 parameters the *setcookie method* in the example above accepts are:

1. The name of the cookie to store.

2. The value of the cookie.

3. How long (in seconds) the cookie will remain on the web client's cookie.txt, before it is deleted.

In addition to these 3 parameters, the method accepts optional restriction parameters as a hash: The `domain => $domain` parameter allows you to set a domain or host name where the cookie may be transmitted to, the `path => $path` parameter restricts the cookie to a URI path and the `secure => $bool` parameter makes sure the cookie is only transmitted when the browser accesses a secure (encrypted, HTTPS) website.

So this more complex example sets a cookie named "person" with value "John Q. Public" that is valid for 2 minutes (120 seconds) and is transmitted back to every host under the "foo.bar" domain (i.e. to "img.foo.bar" as well as to "www.foo.bar"), but only if the requested resource is under the "/private/data/" path and the connection is SSL secured ("https://..."):

```
<%
 $cgi->setcookie('person','John Q. Public',120,
   domain => '*.foo.bar',
   path   => '/private/data/',
   secure => 1
  );
%>
```

If the domain, path and secure options were not set, the cookie would only be transmitted to the host that set it, but regardless to which URI path or whether the data is sent over a secure channel.

Receiving HTTP File Uploads

PSP::CGI also supports multipart/form-data encoded HTTP transports, usually utilized for uploading files to a server using the HTTP and HTTPS protocols. A XHTML form providing a HTTP file upload for JPEG images looks like this:

```
 <html>
  <body>
   <form action="upload.psp" enctype="multipart/form-data"
method="post">
    <p>
     <input type="file" name="image" accept="image/jpeg" /><br />
     <input type="submit" value="Upload!" />
```

```
    </p>
   </form>
  </body>
 </html>
```

To access file uploads, a PSP::CGI object offers two methods: The *uploads method* returns an array of the names of all file uploads, the *upload method* gives you access to the data uploaded.

This example prints the filename and the size of an uploaded file and saves its content to a file on the webserver.

```
 <html>
  <body>
 <%
  my $fh;
  my $file = $cgi->upload('image');
  if(defined($file)){
 %>
   <table>
    <tr>
     <td>Filename:</td>
     <td><%= $file->{filename} %></td>
    </tr>
    <tr>
     <td>Size:</td>
     <td><%= $file->{size} %></td>
    </tr>
    <tr>
     <td colspan="2"><a href="image.jpg">show</a></td>
    </tr>
   </table>
 <%
   open($fh,'>','image.jpg')
    && do{
     print($fh $file->{content});
     close($fh);
    };
  }
  else{
 %>
   <p>
    You did not upload a file. Repeat.
   </p>
 <%
  }
 %>
  </body>
 </html>
```

Like for the *param* and *cookie*, it is also possible to access file uploads directly through `$cgi->upload->{$key}`.

For more information on HTTP file uploads, see *"upload method" in PSP::CGI*.

Custom HTTP Headers

Finally, another important task of PSP::CGI objects is to offer the possibility of defining custom HTTP headers. Popular headers to set/change are "Content-Type:" for defining a content type other than "text/html", for example "image/jpeg" for JPEG images or "text/plain" for plain text output, and the "Status:" header to generate HTTP status messages or i.e. in conjunction with the "Location:" header to create HTTP redirects.

This example uses the *setheader method* to set the "Content-Type:" header, also setting the character set of the web site to Unicode/UTF-8:

```
<%
 $cgi->setheader('Content-Type' => 'text/html; charset=utf-8');
%>
```

Another example creates a "HTTP 302 Moved Temporarily" status message, commonly called a "HTTP redirect", telling the web browser to download the requested data from another/a new location:

```
<%
 $cgi->setheader(
   Status   => '302 Moved Temporarily',
   Location => 'http://foo/bar/blah.html'
  );
%>
```

Especially for HTTP status messages, PSP provides the *PSP::CGI::HTTP module* that provides a much easier and cleaner way for generating such messages. The module will be discussed later.

CGI Parameter Generation Helpers

The *PSP::CGI::Generate module* exports some functions that are to help you to simplify CGI parameter handover.

Generating URI Parameters

The most frequent way of handing over CGI parameters is in URIs, for example in (X)HTML links like this:

```
<a href="http://foo.bar/foobar.psp?a=foo&b=bar">click here</a>
```

Creating such an URI is often fiddly, first of all because data usually requires special encoding. Functions for URI encoding are provided by PSP, in particular by the *encodeHttp function* from the *PSP::CGI::Transform module*, but creating such code is still bugging and inflexible::

```
<%
 use PSP::CGI::Transform;
%>
 <a href="http://foo.bar/foobar.psp?a=<%= encodeHttp($foo) %>&b=<%=
encodeHttp($bar) %>">click here</a>
```

But the *generateGet function* really eases and makes concise this task:

```
<%
 use PSP::CGI::Generate;
%>
 <a href="http://foo.bar/foobar.psp?<%= generateGet(a => $foo, b =>
$bar) %>">click here</a>
```

So the last one does exactly the same as the example given before, but it generates the HTTP GET string directly from a given hash, also taking the work of encoding both keys and values off you.

This function becomes even more useful if your data to be transmitted is already stored in a hash, referenced hash or anonymous hash, as the function accepts all of these types. You may even give more than one hash at once, but mind to not mix native hashs and referenced hashs.

Both examples are correct:

```
<%
 use PSP::CGI::Generate;
 my %hash = (
   a => 1,
   b => 2,
   c => 3
  );
%>
<a href="getthis.psp?<%= generateGet(d => 4, %hash, e => 5) %>">
 Click. Click! CLICK!!!
</a>
```

```
<%
 use PSP::CGI::Generate;
 my $hash = {
   a => 1,
   b => 2,
   c => 3
  };
%>
<a href="getthis.psp?<%= generateGet({d => 4}, $hash, {e => 5}) %>">
 Click. Click! CLICK!!!
</a>
```

But this would be completely wrong:

```
<%
 use PSP::CGI::Generate;
 my %hash = (
   a => 1
  );
 my $hash = {
   b => 2,
   c => 3
  };
%>
 <a href="getthis.psp?<%= generateGet($hash, %hash, {d => 4}, e => 5)
%>">
  Click. Click! CLICK!!!
 </a>
```

If you have both ordinary and referenced/anonymous hashs and want to mix it, you have either to convert the referenced and anonymous ones to ordinary ones, or you reference the ordinary hashs. Because of performance and stack memory reasons, you are advised to use the latter one. The last example corrected in such a way now looks like this:

```
<%
```

```
 use PSP::CGI::Generate;
 my %hash = (
   a => 1
  );
 my $hash = {
   b => 2,
   c => 3
  };
%>
<a href="getthis.psp?<%= generateGet($hash, \%hash, {d => 4, e =>
5}) %>">
 Click. Click! CLICK!!!
</a>
```

See the *generateGet function* for details.

Generating XHTML Hidden Forms

Another frequently utilized way for transmitting data is the use of so called hidden inputs in (X)HTML forms. An example:

```
<form action="manageitem.psp" method="post">
 <p>
  Enter your password if you really want to delete this item:<br />
  <input type="password" name="pass" />
  <input type="submit" value="Kill it!" />
  <input type="hidden" name="do" value="deleteItem" />
  <input type="hidden" name="item" value="1234" />
 </p>
</form>
```

Creating such a form is already really easy as the PrintSecureOpenTag ("<%") does the (X)HTML encoding for you, i.e.

```
<form action="manageitem.psp" method="post">
 <p>
  Enter your password if you really want to delete this item:<br />
  <input type="password" name="pass" />
  <input type="submit" value="Kill it!" />
  <input type="hidden" name="do" value="deleteItem" />
  <input type="hidden" name="item" value="<%= $itemID %>" />
 </p>
</form>
```

but using the *generateForm function* it becomes even easier and clearly arranged:

```
<%
 use PSP::CGI::Generate;
%>
<form action="manageitem.psp" method="post">
 <p>
  Enter your password if you really want to delete this item:<br />
  <input type="password" name="pass" />
  <input type="submit" value="Kill it!" />
  <%= generateForm(do => 'deleteItem', item => $itemID) %>
 </p>
</form>
```

Regarding hashs and referenced/anonymous hashs, this function accepts the same parameters as the *generateGet function* described above.

See the *generateForm function* for details.

AUTHOR

Veit Wahlich

E-Mail: cru |at| zodia |dot| de

WWW: *http://home.ircnet.de/cru/*

VERSION

v0.7 Wednesday, 18 January 2006

COPYRIGHT/LICENSE