HOW TO WRITE A WRAPPER

APAT v1.3

Wrappers are the only hard coded scripts in the entire APAT system. The only requirement for the program is to read and write XML.

Wrappers can be the scripts that run specified local servers or the specified servers over the web. The wrappers that run the web servers need LWP::UserAgent package.

1. Wrappers used to run web servers:

   These wrappers mainly need to handle four important aspects namely,

   a) Getting the XML input data
   b) Passing the required data to the web server
   c) Parsing the output from the web server
   d) Writing XML output containing selected data

2. Wrappers used to run Local server:

   These wrappers mainly need to handle four important aspects namely,

   a) Getting the XML input data
   b) Passing the required data to the local server
   c) Parsing the output from the local server
   d) Writing XML output containing selected data

1. Wrappers used to run web servers:

a) Getting the XML input data:

This can be done using normal programming techniques of pattern matching to identify the required tags and then getting the data. One of the standard ways of parsing XML input data is to use a perl module called XML::DOM designed specifically for parsing. It enables pulling out the required data in a hierarchical fashion whether it is an 'attribute' or enclosed between start<tag> and end</tag> tags by using inbuilt functions like getElementsByTagNam, getAttribute, getFirstChild->getNodeValue.

Example:

Input XMLfile:

```
<input>
   <sequenceid>sp|P17261|ERS1_YEAST Transmembrane protein ERS1 (ERD suppressor)
- Saccharomyces cerevisiae (Baker's yeast).
   </sequenceid>
   <sequence>MVSLDDILGIVYVTSWSISMYPPIITNWRHKSASAISMDFVMLNTAGYSYLVISIFLQLYCWK
             MTGDESDLGRPKLTQFDFWYCLHGCLMNVVLLTQVVAGARIWRFPGKGHRKMNPWYLRILLAS
             LAIFSLLTVQFMYSNYWYDWHNSRTLAYCNNLFLLKISMSLIKYIPQVTHNSTRKSMDCFPIQ
             GVFLDVTGGIASLLQLIWQLSNDQGFSLDTFVTNFGKVGLSMVTLIFNFIFIMQWFVYRSRGH
             DLASEYPL
   </sequence>
   <emailaddress>s.v.v.deevi@rdg.ac.uk</emailaddress>
   <parameter server='targetp' param='origin' value='non-plant' />
   <parameter server='psort' param='origin' value='yeast' />
</input>
```

PERL CODE:

```
use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile (@ARGV[0]);
```

Get the data within input tag.

```
foreach my $input ($doc->getElementsByTagName("input"))
{
```

Move to next level(sub-tag) and pick up the data in sequenceid tag.

```
    $sequenceidtag = $input->getElementsByTagName("sequenceid")->item(0);
    $sequenceid = $sequenceidtag->getFirstChild->getNodeValue;
```

Move to another sub-tag of input tag called sequence tag and get the data.

```
    $sequencetag = $input->getElementsByTagName("sequence")->item(0);
    $sequence = $sequencetag->getFirstChild->getNodeValue;
```

Move to another sub-tag of input tag called emailaddress tag and grab the data.

```
    $emailaddresstag = $input->getElementsByTagName("emailaddress")->item(0);
    $emailaddress = $emailaddresstag->getFirstChild->getNodeValue;
```

Move to another sub-tag of input tag called parameter tag by using foreach loop.

```
        foreach $parameter($input->getElementsByTagName("parameter"))
        {
```

Get the server name from the attribute 'server'.

```
            $server = $parameter->getAttribute("server");
```

Check for the required server using if loop and grab the origin data within the other attribute 'value'.

```
            if($server eq "psort")
            {
                my $param  = $parameter->getAttribute("param");
                if($param eq "origin")
                {
                  $origin = $parameter->getAttribute("value");
                }
            }
        }
    }
```

b) Passing the required data to the web server:

Now the data present within the variables is passed onto specific webserver which then runs the tool with the provided data and sends back the output from the tool.

If required, proxy server should be specified with user/password. As it is not a good practice to place the username and password in a program it can be retrieved from an environment variable.

So if an environment variable WEBPROXY is used for the web proxy, then the .bashrc should include:

```
   export WEBPROXY="user:password@wwwcache.myservername:port"
```

where you substitute your username, password, myservername[example: rdg.ac.uk] and port[example: 8080] as required.

Note:

i)The spacing is important - there must be no spaces around the = sign.
ii)The commands in .bashrc file don't get executed until you run the .bashrc script - i.e. need to open a new shell.
iii)Any scripts that contain your password need to be protected so they can't be read by anyone else:
`chmod og-rx filename`

```
    # Specify proxy server (with user/password) if required
    if(defined($ENV{WEBPROXY}))
    {
      $webproxy = $ENV{WEBPROXY};
    }
    else
    {
      $webproxy = '';
    }
```

Specify the URL for the CGI script to be accessed. For example, to access NetPhos:

```
    $url = "http://www.cbs.dtu.dk/cgi-bin/nph-webface";
```

The submission web page needs to be examined for the required data to be sent to the CGI script and the order in which they should go in.

Note:

For some reason some servers need the configfile to come first as in this case.

```
    $post = "configfile=/usr/opt/www/pub/CBS/services/NetPhos-
2.0/NetPhos.cf&seqpaste=$seq&tyrosine=ps&serine=ps&threonine=ps";
```

A user agent is to be created for posting the request.

```
    $ua = CreateUserAgent($webproxy);

    ########################################################################
    sub CreateUserAgent
    {
         my($webproxy) = @_;

         my($ua);
```

A new LWP::UserAgent is created with a webbrowser like Mozilla.

Header needs to include the line : Use LWP::UserAgent;

```
         $ua = LWP::UserAgent->new;
         $ua->agent('Mozilla/5.0');
```

If proxy is required, provide it with details

```
         if(length($webproxy))
         {
            $ua->proxy(['http', 'ftp'] => $webproxy);
         }
         return($ua);
    }

    ########################################################################
```

After creating the user agent, the request needs to be posted to the URL.

```
    $req = CreatePostRequest($url, $post);

    ########################################################################
    sub CreatePostRequest
    {
        my($url, $params) = @_;
        my($req);
```

The URL is posted in the webbrowser

```
        $req = HTTP::Request->new(POST => $url);
        $req->content_type('application/x-www-form-urlencoded');
    #   $req->content_type('multipart/form-data');
```

The required data for run parameters of the tool are also posted

```
        $req->content($params);

        return($req);
    }

    ########################################################################
```

After posting a request to the URL along with data for the parameters the output from the tool is to be collected.

```
 $result = GetContent($ua, $req);
```

Redirection :

Some servers return redirection page instead of results. The page returns a 'wait' page which eventually redirects itself to the results once they are ready. Modification of the code is needed so that it sits in a loop until the results are available and then grabs them.

Grab this URL out of the returned page.

```
 $url = GrabRedirect($result);
```

```
########################################################################
sub GrabRedirect
{
    my($html) = @_;
```

Grab the URL on the wait page through pattern matching methods.

```
    $html =~ /location\.replace\(\"(.*?)\"\)/;
    return($1);
}

########################################################################
```

So we start a loop...

```
    # Iterate while the URL contains 'wait'
    do
    {
```

Where we create a request using this new URL and get the page which is of the same format but will contain a redirect either to the wait page again or to the final results page

```
        $req = CreateGetRequest($url);
```

Post the new URL

```
    $result = GetContent($ua, $req);
```

Grab the URL

```
    $url = GrabRedirect($result);
```

Just to be polite we sleep for one second so we don't keep hammering the server

```
        sleep 1;
```

and we keep looping while the URL contains the word 'wait'

```
    }   while($url =~ /wait/);
```

We have now got out final results URL so we grab that page

```
    # The URL is now the one for the results page
    $req = CreateGetRequest($url);

########################################################################
sub CreateGetRequest
```

```
{
    my($url) = @_;
    my($req);
```

Gets the new URL

```
    $req = HTTP::Request->new('GET',$url);
    return($req);
}

##########################################################################

 $result = GetContent($ua, $req);
```

The url containing the results is then passed into $link as it is.

```
 $link = $url;
```

As the url might contain '&' character that needs to be written in a different way for HTML to understand, it should be substituted by '&amp;'.

```
 $link =~ s/&/&amp;/g;
```

and we are now ready to return the results.

The results are obtained into a variable called $result which is then passed onto the parsing subroutine.

c) Parsing the output from the web server

Because most of the information presented on the output pages of many tools is not very significant, we need to grab only the information that might be of great significance to a biologist. So we parse the output page and pullout the useful bits only.

```
sub parse
{
   my @data =@_;
   my ($i,$in,@fields,@score);
```

Start a loop that runs until(blank line) there is data in the output page.

```
   for($i=0; $i<@data; $i++)
   {
```

Read the data line by line.

```
      $_ = $data[$i];
```

If we want to get something that follows a header, say for example the header starts with Name in the beginning of line, then we flag that as $in=1 and move to the next line.

```
      if(/^Name/)
      {
          $i++;
          $in=1;
      }
```

Similarly we flag something as end, say for example the end is marked by a series of continous '_' signs at the start of the line, then we flag that as $in=0

```
      elsif(/^______/)
      {
         $in=0;
```

```
        }
```

When the start flag is found, we enter a loop to pull the data.

```
        elsif($in)
        {
           s/^\s+//;
```

Enter if the line is not empty

```
           if(length())
           {
```

If we want to split the various columns into different fields and get the values of required fields only

```
              @fields=split();
```

If $field[1] is residue number and $field[3] is some score we are interested in storing in an array marked by the residue numbers

```
               $score[$fields[1]-1]=$fields[3];
           }
        }

     }
```

The parsed data is returned.

```
     return(@score);
  }
```

d) Writing XML output containing selected data

The parsed data is now printed out using the designed XML tags.

For example:

A bunch of lines can be printed by using a format as below. 'print ‘ statement followed by an End Of File character (can be anything) and semicolon indicates the starting point of printable lines. When the End Of File character is printed again, then it indicates the end of printable region.

Note:

Closing End Of File character should be at the very beginning of the line without any trailing spaces.

```
     print <<__EOF;
    <result program='NetPhos' version='2.0'>
       <function>Protein Phosphorylation sites Prediction</function>
       <info href='http://www.cbs.dtu.dk/services/NetPhos/'>NetPhos Web
Server</info>
       <run>
          <params>
             <param name = 'Serine' value = 'Checked'/>
           <param name = 'Threonine' value = 'Checked'/>
           <param name = 'Tyrosine' value = 'Checked'/>
           <param name = 'Generate Graphics' value = 'unChecked'/>
           <param name = 'Threshold' value = '0.500'/>
          </params>
        <date>$dt</date>
       </run>
       <predictions>
          <link href='$link'>Actual prediction(native, unparsed form)- available
only for a limited time</link>
```

```
          <perres-number name = 'P-score' clrmin = '0.0' clrmax = '1.0' graph='1'
graphtype='bars'>
__EOF
```

When some programming chunk is to be added, then the bulk printing is ended.

```
                $count=1;
                foreach $val(@score)
                {
                  $k = $count;
```

As there will be multiple value-perres tags they are conveniently looped around the array carrying the values.

```
                printf "              <value-perres residue='$k'>%f</value-
perres>\n",$val;

        <threshold>
           <description>P-scores greater than 0.5 are considered as positive
predictions
           </description>
__EOF
```

and so on……….

2. Wrappers used to run Local server:

a) Getting the XML input data:

Same as in webserver's context

b) Passing the required data to the local server:

This is much easier and better way of doing it. If the source code of a tool is open and free to download, it's always advisable to have it locally because of the following reasons.

i) Local programs run faster as they avoid network traffic.
ii) They don't need to use complex ways of posting the input.
iii) They can avoid the risky screen scraping techniques which fail to work if the format of webpages is changed.

For example:

After getting the input data , it can be written to a file which can then be conveniently passed onto the program on the command line.

```
         my($ipf) =@_;
         my($inputfile,$outputfile);

         open(FILE,">inputfile") || die "Can't write inputfile";
         print FILE $ipf;
         close FILE;

         $inputfile = "inputfile";
```

Run the tool(program) on the command line by using backticks ``.

```
$output = `~/psipred/runpsipred $inputfile`;
```

Now the output is in a variable called $output.

c) Parsing the output from the local server:

Same as in webserver's context.

d) Writing XML output containing selected data:

Same as in webserver's context.