NAME

yaws_api − api available to yaws web server programmers

SYNOPSIS

yaws_api:Function(...)

DESCRIPTION

This is the api available to yaws web server programmers. The Erlang module yaws_api contains a wide variety of functions that can be used inside yaws pages.

Each chunk of yaws code is executed while the yaws page is being delivered from the server. We give a very simple example here to show the basic idea. Imagine the following HTML code:

```
<html>
<body>

<h1> Header 1</h1>

<erl>
out(Arg) ->
   {html, "<p> Insert this text into the document"}.
</erl>

</body>
</html>
```

The **out(Arg)** function is supplied one argument, an #arg{} structure. We have the following relevant record definitions:

```
-record(arg, {
          clisock,        %% the socket leading to the peer client
          client_ip_port, %% {ClientIp, ClientPort} tuple
          headers,        %% headers
          req,            %% request
          clidata,        %% The client data (as a binary in POST requests)
          server_path,    %% The normalized server path
          querydata,      %% Was the URL on the form of ...?query (GET reqs)
          appmoddata,     %% the remainder of the path up to the query
          docroot,        %% where's the data
          fullpath,       %% full path to yaws file
          cont,                 %% Continuation for chunked multipart uploads
          state,          %% State for use by users of the out/1 callback
          pid,            %% pid of the yaws worker process
          opaque,         %% useful to pass static data
          appmod_prepath, %% path in front of: <appmod><appmoddata>
          pathinfo        %% Set to 'd/e' when calling c.yaws for the request
                          %% http://some.host/a/b/c.yaws/d/e
         }).
```

The headers argument is also a record:

```
-record(headers, {
          connection,
          accept,
          host,
          if_modified_since,
          if_match,
          if_none_match,
          if_range,
          if_unmodified_since,
          range,
          referer,
          user_agent,
          accept_ranges,
          cookie = [],
          keep_alive,
          content_length,
          authorization,
          other = []   %% misc other headers
         }).
```

it likes. We have the following functions to aid that generation.

API

ssi(DocRoot, ListOfFiles)
Server side include. Just include the files as is in the document. The files will **not** be parsed and searched for <erl> tags.

pre_ssi_files(DocRoot, ListOfFiles) ->
Server side include of pre indented code. The data in Files will be included but contained in a <pre> tag. The data will be htmlized.

pre_ssi_string(String)
Include htmlized content from String.

f(Fmt, Args)
The equivalent of io_lib:format/2. This function is automatically -included in all erlang code which is a part of a yaws page.

htmlize(Binary | List | Char)
Htmlize an IO list object.

setcookie(Name, Value, [Path, [Expire, [Domain , [Secure]]]])
Sets a cookie to the browser.

find_cookie_val(Cookie, Header)
This function can be used to search for a cookie that was previously set by **setcookie/2-6**. For example if we set a cookie as **yaws_api:setcookie("sid",SomeRandomSid)** , then on subsequent requests from the browser we can call: **find_cookie("sid",(Arg#arg.headers)#headers.cookie)**

The function returns [] if no cookie was found, otherwise the actual cookie is returned as a string.

redirect(Url
This function generates a redirect to the browser. It will clear any previously set headers. So to generate a redirect **and** set a cookie, we need to set the cookie after the redirect as in:

```
out(Arg) ->
  ... do some stuff

  Ret = [{redirect, "http://www.somewhere.com"},
         setcookie("sid", Random)
        ].
```

redirect_self(Arg)
If we want to issue a redirect to ourselves, this function is useful. It returns a record *#redir_self{}* defined in *yaws_api.hrl*. The record contains fields to construct a URL to ourselves.

```
-record(redir_self, {
        host,        %% string() - our own host
        scheme,      %% http | https
        scheme_str,  %% "https://"  | "http://"
        port,        %% integer()  - our own port
        port_str     %% "" | ":<int>" - the optional port part
                     %%                to append to the url
       }).
```

get_line(String)
This function is convenient when getting \r\n terminated lines from a stream of data. It returns:

{line, Line, Tail} or **{lastline, Line, Tail}**

The function handles multilines as defined in e.g. SMTP or HTTP

mime_type(FileName)
Returns the mime type as defined by the extension of FileName

stream_chunk_deliver(YawsPid, Data)
When a yaws function needs to deliver chunks of data which it gets from a process. The other process can call this function to deliver these chunks. It requires the **out/1** function to return the value **{streamcontent, MimeType, FirstChunk}** to work. YawsPid is the process identifier of the

yaws process delivering the original .yaws file. That is self() in the yaws code. The Pid must typically be passed (somehow) to the producer of the stream.

stream_chunk_deliver_blocking(YawsPid, Data)
A synchronous version of the above function. This synchronous version must always be used when the producer of the stream is faster than the consumer. This is usually the case since the client is the WWW browser.

stream_chunk_end(YawsPid)
When the process discussed above is done delivering data, it must call this function to let the yaws content delivering process finish up the HTTP transaction.

stream_process_deliver(Socket, IoList)
Yaws allows application processes to deliver data directly to the client. The application tells yaws about such a process by returning **{streamcontent_from_pid, MimeType, Pid}** from its **out/1** function. In this case, *Pid* uses the **stream_process_deliver/2** function to deliver data to the client. The application gets *Socket* from *Arg#arg.clisock*, and *IoList* is the data to be sent to the client.

stream_process_deliver_chunk(Socket, IoList)
Same as above but delivers *IoList* using HTTP chunked transfer format. *IoList* must have a size greater than zero. The application process delivering the data will have had to have make sure that the HTTP headers of the response indicate chunked transfer mode, either by ensuring no Content-Length header is set or by specifically setting the Transfer-Encoding header to chunked.

stream_process_deliver_final_chunk(Socket, IoList)
If the application process delivering data to the client uses chunked transfer mode, it must call this to deliver the final chunk of the transfer. This tells yaws to create a special final chunk in the format required by the HTTP specification (RFC 2616). *IoList* may be empty, but if its size is greater than zero, that data will be sent as a separate chunk before the final chunk.

stream_process_end(Socket, YawsPid)
Application processes delivering data directly to clients must call this function to inform yaws that they've finished using *Socket*. The *YawsPid* argument will have been passed to the process earlier when yaws sent it a message telling it to proceed with data delivery. Yaws expects *Socket* to be open.

stream_process_end(closed, YawsPid)
Same as the previous function but the application calls this if it closes the client socket as part of its data delivery process. This allows yaws to continue without assuming the socket is still open and encountering errors due to that assumption. The *YawsPid* argument will have been passed to the application process earlier when yaws sent it a message telling it to proceed with data delivery.

parse_query(Arg)
This function will parse the query part of the URL. It will return a {Key, Value} list of the items supplied in the query part of the URL.

queryvar(Arg, VarName)
This function is automatically included from yaws_api in all
.yaws pages. It is used to search for a variable in the querypart of the url. Returns {ok, Val} or undefined. If a variable is defined multiple times, the function may also return *{Val1, ...}*.

parse_post(Arg)

This function will parse the POST data as supplied from the browser. It will return a {Key, Value} list of the items set by the browser.

postvar(Arg, VarName)

This function is automatically included from yaws_api in all .yaws pages. It is used to search for a variable in the POSTed data from the client. Returns {ok, Val} or undefined. If a variable is defined multiple times, the function may also return *{Val1, ...}*.

getvar(Arg, VarName)

This function looks at the HTTP request method from the client and invokes postvar/2 if it is a POST from the client and queryvar/2 if it is a GET request from the client.

parse_multipart_post(Arg)

If the browser has set the Content-Type header to the value "multipart/form-data", which is the case when the browser wants to upload a file to the server the following happens:

If the function returns **{result, Res}** no more data will come from the browser.

If the function returns **{cont, Cont, Res}** the browser will supply more data. (The file was to big to come in one read)

This indicates that there is more data to come and the out/1 function should return {get_more, Cont, User_state} where User_state might usefully be a File Descriptor. The Res value is a list of either: **{head, {Name, Headers}} | {part_body, Binary} | {body, Binary}**

The function returns **{error, Reason}** when an error occurred during the parsing.

Example usage could be:

```
<erl>

out(A) ->
       case yaws_api:parse_multipart_post(A) of
            {cont, Cont, Res} ->
                   St = handle_res(A, Res),
                   {get_more, Cont, St};
            {result, Res} ->
                   handle_res(A, Res),
                   {html, f("<pre>Done </pre>",[])};
            {error, Reason} ->
                   {html, f("An error occured: ~p", [Reason])}
       end.

handle_res(A, [{head, {Name, _Hdrs}}|T]) ->
    io:format("head:~p~n",[Name]),
    handle_res(A, T);
handle_res(A, [{part_body, Data}|T]) ->
    io:format("part_body:~p~n",[Data]),
    handle_res(A, T);
handle_res(A, [{body, Data}|T]) ->
    io:format("body:~p~n",[Data]),
    handle_res(A, T);
handle_res(A, []) ->
    io:format("End_res~n").

</erl>
```

new_cookie_session(Opaque)
Create a new cookie based session, the yaws system will set the cookie. The new random generated cookie is returned from this function. The Opaque argument will typically contain user data such as user name and password

new_cookie_session(Opaque, TTL)
As above, but allows to set a session specific time-out value, overriding the system specified time-out value.

new_cookie_session(Opaque, TTL, CleanupPid)
As above, but also sends a message *{yaws_session_end, Reason, Cookie, Opaque}* to the provided CleanuPid where Reason can be either of *timeout* or *normal*. The *Cookie* is the HTTP cookie as returned by *new_session()* and the Opaque is the user provided Opaque parameter to *new_session()*. The purpose of the feature is to cleanup resources assigned to the session.

cookieval_to_opaque(CookieVal)

print_cookie_sessions()

replace_cookie_session(Cookie, NewOpaque)

delete_cookie_session(Cookie)

setconf(Gconf, Groups)
This function is intended for embedded mode in yaws. It makes it possible to load a yaws configuration from another data source than /etc/yaws.conf, such as a database. If yaws is started with the environment *{embedded, true}*, yaws will start with an empty default configuration, and wait for some other program to execute a *setconf/2* The Gconf is a *#gconf{}* record and the Group variable is a list of lists of *#sconf{}* records. Each sublist must contain *#sconf{}* records with the same IP/Port listen address. To create a suitable initial #gconf{} record see the code in yaws_config:make_default_gconf/2. Especially the *yaws_dir* parameter is important to get right.

url_decode(Str)
Decode url-encoded string. A URL encoded string is a string where all alfa numeric characters and the the character _ are preserved and all other characters are encode as "%XY" where X and Y are the hex values of the least respective most significant 4 bits in the 8 bit character.

url_encode(Str)
Url-encodes a string. All URLs in HTML documents must be URL encoded.

reformat_header(H)
Returns a list of reformatted header values from a #header{} record. The return list is suitable for retransmit.

request_url(ARG)
Return the url as requested by the client. Return value is a #url{} record as defined in yaws_api.hrl

parse_url(Str)
Parse URL in a string, returns a #url record

format_url(UrlRecord)
Takes a #url record a formats the Url as a string

call_cgi(Arg, Scriptfilename)
Calls an executable CGI script, given by its full path. Used to make '.yaws' wrappers for CGI programs. This function usually returns *streamcontent*.

call_cgi(Arg, Exefilename, Scriptfilename)
Like before, but calls *Exefilename* to handle the script. The file name of the script is handed to the executable via a CGI meta variable.

call_fcgi_responder(Arg)
Calls a FastCGI responder. The address and port of the FastCGI application server are taken from the server configuration (see yaws.conf). Used to make '.yaws' wrappers for FastCGI responders. Returns the same return values as out/1 (see below).

call_fcgi_responder(Arg, Options)
Same as above, but Options overrides the defaults from the server configuration:

Options = [Option]
Option -- one of the following:

{app_server_host, string() | ip_address()} The hostname or the IP address of the FastCGI application server.

{app_server_port, 0..65535} The TCP port number of the FastCGI application server.

{path_info, string()} Override default pathinfo in Arg#arg.pathinfo.

{extra_env, ExtraEnv} Override default pathinfo in Arg#arg.pathinfo.

ExtraEnv = [Var]
Var = {Name, Value}

Name = string()
Value = string()

{trace_protocol, boolean()} Enable or disable tracing of FastCGI protocol messages as info log messages.

{log_app_error, boolean()} Enable or disable logging of application error messages: output to stderr and non-zero exit value.

call_fcgi_authorizer(Arg) -> {allowed, Out} | {denied, Out}

Calls a FastCGI authorizer. The address and port of the FastCGI application server are taken from the server configuration (see yaws.conf). Used to make ‘.yaws’ wrappers for FastCGI authorizers. Variables contains the values of the variables returned by the FastCGI application server in the "Variable-XXX: YYY" headers.

If access is denied, Out contains the complete response returned by the FastCGI application server. This response is typically returned as-is to the HTTP client.

If access is allowed, Out contains the response returned by the FastCGI application server minus the body (i.e. minus the content) which should be ignored per the FastCGI specification. This response is typically not returned to the HTTP client. The calling application module may wish to inspect the response, for example by extracting variables (see fcgi_extract_variables below) or by inspecting the headers returned by the FastCGI application server.

Out -- See return values for out/1 below

call_fcgi_authorizer(Arg, Options) -> {allowed, Out} | {denied, Out}

Same as above, but Options overrides the defaults from the server configuration. See call_fcgi_responder/2 above for a description of Options.

fcgi_extract_variables(Out) -> [{Name, Value}]

Extracts the environment variables from a FastCGI authorizer response by looking for headers of the form "Variable-Name: Value".

Name = string() -- The name of the variable (the "Variable-" prefix
has already been removed).
Value = string() -- The value of the variable.

dir_listing(Arg)

Perform a directory listing. Can be used in special directories when we don’t want to turn on dir listings for the entire server. Always returns ok.

RETURN VALUES from out/1

The out/1 function can return different values to control the behavior of the server.

{html, DeepList}

This assumes that DeepList is formatted HTML code. The code will be inserted in the page.

{ehtml|exhtml, Term}

This will transform the erlang term Term into a stream of HTML content. The exhtml variant transforms into strict XHTML code. The basic syntax of Term is

```
EHTML = [EHTML] | {Tag, Attrs, Body} | {Tag, Attrs} | {Tag} |
       binary() | character()
Tag     = atom()
Attrs = [{Key, Value}]  or {EventTag, {jscall, FunName, [Args]}}
Key     = atom()
Value = string()
Body  = EHTML
```

For example, *{p, [], "Howdy"}* expands into "<p>Howdy</p>" and

```
{form, [{action, "a.yaws"}],
   {input, [{type,text}]}}
```

expands into

```
<form action="a.yaws"
  <input type="text">
</form>
```

It may be more convenient to generate erlang tuples than plain html code.

{content, MimeType, Content}

This function will make the web server generate different content than HTML. This return value is only allowed in a yaws file which has only one <erl> </erl> part and no html parts at all.

{streamcontent, MimeType, FirstChunk}

This return value plays the same role as the *content* return value above.

However it makes it possible to stream data to the client if the yaws code doesn't have access to all the data in one go. (Typically if a file is very large or if data arrives from back end servers on the network.

{streamcontent_with_timeout, MimeType, FirstChunk, Timeout}

Similar to above, but with an explicit timeout. The default timeout is 30 secs. I.e if the application fails to deliver data to the Yaws process, the streaming will stop. This is often not the desired behaviour in Comet/Ajax applications. It's possible to provide 'infinity' as timeout.

{streamcontent_from_pid, MimeType, Pid}

This return value is similar to the *streamcontent* return value above.

However it makes it possible to stream data to the client directly from an application process to the socket. This approach can be useful for applications that employ long-polling (Comet) techniques, for example, and for applications wanting to avoid buffering data or avoid HTTP chunked mode transfer for streamed data.

{streamcontent_with_size, Sz, MimeType, FirstChunk}

This return value is similar to the *streamcontent* return value above.

However it makes it possible to stream data to the client by setting the content length of the response. As the opposite of other ways to stream data, in this case, the response is not chunked encoded.

{header, H}

Accumulates a HTTP header. The trailing CRNL which is supposed to end all HTTP headers must NOT be added. It is added by the server. The following list of headers are given special treatment.

{connection, What}

This sets the Connection: header. If *What* is the special value *"close"*, the connection will be closed once the yaws page is delivered to the client.

{server, What}

Sets the Server: header. By setting this header, the server's signature will be dynamically overloaded.

{location, Url}

Sets the Location: header. This header is typically combined with the *{status, 302}* return value.

{cache_control, What}

Sets the Cache-Control: header.

{expires, What}

Sets the Expires: header.

{date, What}

Sets the Date: header.

{allow, What}

Sets the Allow: header.

{last_modified, What}

Sets the Last-Modified: header.

{etag, What}

Sets the Etag: header.

{set_cookie, Cookie}

Prepends a Set-Cookie: header to the list of previously set Set-Cookie: headers.

{content_range, What}

Sets the Content-Range: header.

{content_type, MimeType}

Sets the Content-Type: header.

{content_encoding, What}

Sets the Content-Encoding: header. If this header is defined, no deflate is performed by Yaws. So you can compress data by yourself.

{content_length, Len}

Normally yaws will ship Yaws pages using Transfer-Encoding: chunked. This is because we generally can't know how long a yaws page will be. If we for some reason want to force a Content-Length: header (and we actually do know the length of the content, we can force yaws to not ship the page chunked.

{transfer_encoding, What}

Sets the Transfer-Encoding: header.

{www_authenticate, What}

Sets the WWW-Authenticate: header.

All other headers must be added using the normal HTTP syntax. Example:

{header, {"My-X-Header", "gadong"}} of *{header, "My-X-Header: gadong"}*

{allheaders, HeaderList}
Will clear all previously accumulated headers and replace them.

{status, Code}
Will set another HTTP status code than 200.

break Will stop processing of any consecutive chunks of erl or html code in the yaws file.

ok Do nothing.

flush Flush remaining data sent by the client.

{redirect, Url}
Erase all previous headers and accumulate a single Location header. Set the status code.

{redirect_local, Path}
Does a redirect to the same Scheme://Host:Port/Path as we currently are executing in.

{get_more, Cont, State}
When we are receiving large POSTs we can return this value and be invoked again when more Data arrives.

{page, Page}
Make Yaws return a different page than the one being requested.

{page, {Options, Page}}
Like the above, but supplying an additional deep list of options. For now, the only type of option is *{header, H}* with the effect of accumulating the HTTP header *H* for page *Page*.

{ssi, File, Delimiter, Bindings}
Server side include File and macro expansion in File. Each occurrence of a string, say "xyz", inside File which is inside Delimiters is replaced with the corresponding value in Bindings.

Example: Delimiter = %%

File contains the string .... %%xyz%% .....

Bindings contain the tuple {"xyz", "Dingbat"}

The occurrence of %%xyz%% in File will be replaced with "Dingbat" in the Server side included output.

The {ssi, File, Delimiter, Bindings} statement can also occur inside a deep ehtml structure.

{bindings, [{Key1, Value2}, {Key2, Value2} .....]}
Establish variable bindings that can be used in the page.

All bindings can then be used in the rest of yaws code (in HTML source and within erl tags). In HTML source %%Key%% is expanded to Value and within erl tags *yaws_api:binding(Key)* can be used to extract Value and *yaws_api:binding_exists(Key)* can be used to check for the existence of a binding.

{yssi, YawsFile}
Include a yaws file. Compile it and expand as if it had occured inline.

[ListOfValues]
It is possible to return a deep list of the above defined return values. Any occurrence of *streamcontent*, *streamcontent_with_timeout*, *streamcontent_with_size*, *streamcontent_from_pid*, *get_more*, *page* or *break* in this list is legal only if it is the last position of the list. If not, remaining values in the list are ignored.

AUTHOR

Written by Claes Wikstrom

SEE ALSO

yaws.conf(5) **erl**(1)