Yaws - Yet Another Web Server

Claes Wikstrom
klacke@hyber.org

August 18, 2024

Contents

Chapter 1

Introduction

Yaws is an Erlang web server. It's written in Erlang and it uses Erlang as its embedded language similar to PHP in Apache or Java in Tomcat.

The advantages of Erlang as an embedded web page language as opposed to Java or PHP are many.

- Speed - Using Erlang for both implementing the web server itself as well as embedded script language gives excellent dynamic page generation performance.
- Beauty - Well this is subjective
- Scalability - due to the lightweight processes of Erlang, Yaws is able to handle a very large number of concurrent connections

Yaws has a wide feature set; it supports:

- HTTP 1.0 and HTTP 1.1
- Static content page delivery
- Dynamic content generation using embedded Erlang code in the HTML pages
- NCSA combined/XLF/ELF log format traffic logs
- Virtual hosting with several servers on the same IP address
- Multiple servers on multiple IP addresses
- HTTP tracing for debugging
- An interactive interpreter environment in the Web server for use while developing and debugging a web site

- RAM caching of commonly accessed pages
- Full streaming capabilities of both upload and download of dynamically generated pages
- SSL
- Support for WWW-Authenticated pages
- Support API for cookie based sessions
- Application Modules where virtual directory hierarchies can be made
- Embedded mode
- WebSockets (RFC 6455)
- Long polling (COMET) applications
- Forward and reverse proxying

1.1 Prerequisites

This document requires that the reader:

- Is well acquainted with the ERLANG programming language.
- Understands basic Web technologies.

1.2 A tiny example

We introduce YAWS by help of a tiny example. The web server YAWS serves and delivers static content pages similar to any old web server, except that YAWS does this much faster than most web servers. It's the dynamic pages that makes YAWS interesting. Any page with the suffix ".yaws" is considered a dynamic YAWS page. A YAWS page can contain embedded ERLANG snippets that are executed while the page is being delivered to the WWW browser.

Example 1.1 is the HTML code for a small YAWS page.

It illustrates the basic idea behind YAWS. The HTML code, generally stored in a file ending with a ".yaws" suffix, can contain `<erl>` and `</erl>` tags and inside these tags an ERLANG function called `out/1` gets called and the output of that function is inserted into the HTML document, dynamically.

It is possible to have several chunks of HTML code together with several chunks of ERLANG code in the same YAWS page.

The `Arg` argument supplied to the automatically invoked `out/1` function is an ERLANG record that contains various data which is interesting when generating dynamic pages. For example the HTTP headers which were sent from the WWW client, the actual TCP/IP socket leading to the WWW client. This will be elaborated on thoroughly in later chapters.

The `out/1` function returned the tuple `{html, String}` and `String` gets inserted into the HTML output. There are number of different return values that can be returned from the `out/1` function in order to control the behavior and output from the YAWS web server.

```
<html>

<p> First paragraph

<erl>
out(Arg) ->
    {html, "<p>This string gets inserted into HTML document dynamically"}.
</erl>

<p> And here is some more HTML code

</html>
```

Figure 1.1: Example 1.1

Chapter 2

Compile, Install, Config and Run

This chapter is more of a "Getting started" guide than a full description of the YAWS configuration. YAWS is hosted on Github at `https://github.com/erlyaws/yaws`. This is where the source code resides in a git repository and the latest unreleased version is available via git through the following commands:

```
$ git clone https://github.com/erlyaws/yaws
```

Released version of YAWS are available at `https://github.com/erlyaws/yaws/releases`.

2.0.1 Compile and Install

To compile and install a YAWS release one of the prerequisites is a properly installed ERLANG system. YAWS runs on ERLANGOTP releases 23.0 and newer. Get ERLANG from `http://www.erlang.org/`

Compile and install is straight forward:

```
# cd /usr/local/src
# tar xfz yaws-X.XX.tar.gz
# cd yaws-X.XX
# ./configure && make
# make install
```

The `make` command will compile the YAWS web server with the `erlc` compiler found by the configure script.

- `make install` - will install the executable called `yaws` in `/usr/local/bin/` and a working configuration file in `/usr/local/etc/yaws.conf`

Alternatively, you can compile YAWS with `rebar` as follows:

```
# rebar get-deps compile
```

If you want to build with SOAP support, run the following command:

```
# YAWS_SOAP=1 rebar get-deps compile
```

To create a YAWS release with `reltool`, execute the following command:

```
# rebar generate
```

Because it bundles Erlang/OTP and all of the application's dependencies, the generated release found in `rel/` is standalone and has no external requirements. A future release of `rebar` will allow you to create a slim release that doesn't bundle Erlang/OTP. This is not yet available.

While developing a YAWS site, it's typically most convenient to do a `local` install and run YAWS as a non-privileged user using `--prefix` option of the `configure` script:

```
# ./configure --prefix=/path/to/yaws && make install
# /path/to/yaws/bin/yaws -i
```

2.0.2 Configure

Let's take a look at the config file that gets written after a `local` install in `/home/klacke/yaws/`. The file is `/home/klacke/yaws/etc/yaws/yaws.conf`:

```
# first we have a set of globals

logdir = /home/klacke/yaws/var/log/yaws
ebin_dir = /home/klacke/yaws/lib/yaws/examples/ebin
include_dir = /home/klacke/yaws/lib/yaws/examples/include

...

# and then a set of servers

<server localhost>
        port = 8000
        listen = 127.0.0.1
        docroot = /home/klacke/yaws/var/yaws/www
</server>
```

Figure 2.1: Minimal Local Configuration

The configuration consists of an initial set of global variables that are valid for all defined servers.

The only global directive we need to care about for now is the logdir. YAWS produces a number of log files. We start YAWS interactively as

```
# ~/bin/yaws -i
Erlang (BEAM) emulator version 5.1.2.b2 [source]

Eshell V5.1.2.b2  (abort with ^G)
1>
=INFO REPORT==== 30-Oct-2002::01:38:22 ===
Using config file /home/klacke/yaws/etc/yaws/yaws.conf
=INFO REPORT==== 30-Oct-2002::01:38:22 ===
Listening to 127.0.0.1:8000 for servers ["localhost:8000"]

1>
```

By starting YAWS in interactive mode (using the command switch *-i*) we get a regular ERLANG prompt. This is most convenient when developing YAWS pages. For example we:

- Can dynamically compile and load optional helper modules we need.
- Get all the crash and error reports written directly to the terminal.

The configuration in Example 2.1 defined one HTTP server on address 127.0.0.1:8000 called "localhost". It is important to understand the difference between the name and the address of a server. The name is the expected value in the client HTTP `Host:` header. That is typically the same as the fully-qualified DNS name of the server whereas the address is the actual IP address of the server.

Since YAWS supports virtual hosting with several servers on the same IP address, this matters.

Nevertheless, our server listens to *127.0.0.1:8000* and has the name "localhost", thus the correct URL for this server is `http://localhost:8000`.

The document root (docroot) for the server is a copy of the `www` directory in the YAWS source code distribution. This directory contains a bunch of examples and we should be able to run all those example now on the URL `http://localhost:8000`.

Instead of editing and adding files in the YAWS `www` directory, we create yet another server on the same IP address but a different port number — and in particular a different document root where we can add our own files.

```
# mkdir ~/test
# mkdir ~/test/logs
```

Now change the config so it looks like this:

```
logdir = /home/klacke/test/logs
ebin_dir = /home/klacke/test
include_dir = /home/klacke/test

<server localhost>
        port = 8000
```

```
        listen = 127.0.0.1
        docroot = /home/klacke/yaws/var/yaws/www
</server>

<server localhost>
        port = 8001
        listen = 127.0.0.1
        docroot = /home/klacke/test
</server>
```

We define two servers, one being the original default and a new pointing to a document root in our home directory.

We can now start to add static content in the form of HTML pages, dynamic content in the form of `.yaws` pages or ERLANG `.beam` code that can be used to generate the dynamic content.

The load path will be set so that beam code in the directory `~/test` will be automatically loaded when referenced.

It is best to run YAWS interactively while developing the site. In order to start the YAWS as a daemon, we give the flags:

```
# yaws -D --heart
```

The *-D* or *–daemon* flags instructs YAWS to run as a daemon and the *–heart* flag will start a heartbeat program called heart which restarts the daemon if it should crash or if it stops responding to a regular heartbeat. By default, heart will restart the daemon unless it has already restarted 5 times in 60 seconds or less, in which case it considers the situation fatal and refuses to restart the daemon again. The *-heart-restart=C,T* flag changes the default 5 restarts in 60 seconds to *C* restarts in *T* seconds. For infinite restarts, set both *C* and *T* to 0. This flag also enables the *–heart* flag.

Once started in daemon mode, we have very limited ways of interacting with the daemon. It is possible to query the daemon using:

```
# yaws -S
```

This command produces a simple printout of uptime and number of hits for each configured server.

If we change the configuration, we can HUP the daemon using the command:

```
# yaws -h
```

This will force the daemon to reread the configuration file.

Chapter 3

Static content

YAWS acts very much like any regular web server while delivering static pages. By default YAWS will cache static content in RAM. The caching behavior is controlled by a number of global configuration directives. Since the RAM caching occupies memory, it may be interesting to tweak the default values for the caching directives or even to turn it off completely.

The following configuration directives control the caching behavior

- *max_num_cached_files = Integer*

  YAWS will cache small files such as commonly accessed GIF images in RAM. This directive sets a maximum number on the number of cached files. The default value is 400.

- *max_num_cached_bytes = Integer*

  This directive controls the total amount of RAM which can maximally be used for cached RAM files. The default value is 1000000, 1 megabyte.

- *max_size_cached_file = Integer*

  This directive sets a maximum size on the files that are RAM cached by YAWS. The default value is 8000 bytes, 8 batters.

It may be considered to be confusing, but the numbers specified in the above mentioned cache directives are local to each server. Thus if we have specified `max_num_cached_bytes = 1000000` and have defined 3 servers, we may actually use $3 * 1000000$ bytes.

Chapter 4

Dynamic content

Dynamic content is what YAWS is all about. Most web servers are designed with HTTP and static content in mind whereas YAWS is designed for dynamic pages from the start. Most large sites on the Web today make heavy use of dynamic pages.

4.1 Introduction

When the client `GET`s a page that has a ".yaws" suffix, the YAWS server will read that page from the hard disk and divide it in parts that consist of HTML code and ERLANG code. Each chunk of ERLANG code will be compiled into a module. The chunk of ERLANG code must contain a function `out/1`. If it doesn't the YAWS server will insert a proper error message into the generated HTML output.

When the YAWS server ships a `.yaws` page it will process it chunk by chunk through the `.yaws` file. If it is HTML code, the server will ship that as is, whereas if it is ERLANG code, the YAWS server will invoke the `out/1` function in that code and insert the output of that `out/1` function into the stream of HTML that is being shipped to the client.

YAWS will (of course) cache the result of the compilation and the next time a client requests the same `.yaws` page YAWS will be able to invoke the already-compiled modules directly.

4.2 EHTML

There are two ways to make the `out/1` function generate HTML output. The first and most easy to understand is by returning a tuple `{html, String}` where `String` then is regular HTML data (possibly as a deep list of strings and/or binaries) which will simply be inserted into the output stream. An example:

```
<html>
<h1> Example 1 </h1>

<erl>
out(A) ->
    Headers = A#arg.headers,
    {html, io_lib:format("You say that you're running ~p",
```

```
                          [Headers#headers.user_agent])}.

</erl>

</html>
```

The second way to generate output is by returning a tuple `{ehtml, EHTML}` or `{exhtml, EHTML}`. The exhtml variant generates strict XHTML code. The term `EHTML` must adhere to the following structure:

$$
\begin{aligned}
EHTML = {} & [EHTML]|\{TAG,Attrs,Body\}|\{TAG,Attrs\}|\{TAG\}| \\
& \{Module,Fun,[Args]\}|fun/0| \\
& binary()|character()
\end{aligned}
$$

$$TAG = atom()$$

$$Attrs = [\{HtmlAttribute,Value\}]$$

$$HtmlAttribute = atom()$$

$$
\begin{aligned}
Value = {} & string()|binary()|atom()|integer()|float()| \\
& \{Module,Fun,[Args]\}|fun/0
\end{aligned}
$$

$$Body = EHTML$$

We give an example to show what we mean. The tuple

```
{ehtml, {table, [{bgcolor, grey}],
         [
          {tr, [],
           [
            {td, [], "1"},
            {td, [], "2"},
            {td, [], "3"}
           ]
          },
          {tr, [],
           [{td, [{colspan, "3"}], "444"}]}]}}.
```

expands into the following HTML code:

```
<table bgcolor="grey">
  <tr>
    <td> 1 </td
    <td> 2 </td>
    <td> 3 </td>
  </tr>
  <tr>
    <td colspan="3"> 444 </td>
  </tr>
</table>
```

At a first glance it may appears as if the HTML code is more beautiful than the ERLANG tuple. That may very well be the case from a purely aesthetic point of view. However the ERLANG code has the advantage of being perfectly indented by editors that have syntax support for ERLANG (read Emacs). Furthermore, the ERLANG code is easier to manipulate from an ERLANG program.

Note that ehtml supports function calls as values. Functions can return any legal ehtml value, including other function values. YAWS supports `{M,F,[Args]}` and `fun/0` function value forms.

As an example of some more interesting ehtml we could have an `out/1` function that prints some of the HTTP headers. In the `www` directory of the YAWS source code distribution we have a file called `arg.yaws`. The file demonstrates the `Arg #arg` record parameter which is passed to the `out/1` function.

But before we discuss that code, we describe the `Arg` record in detail.

Here is the `yaws_api.hrl` file which is in included by default in all YAWS files. The `#arg{}` record contains many fields that are useful when processing HTTP request dynamically. We have access to basically all the information associated with the client request such as:

- The actual socket leading back to the HTTP client
- All the HTTP headers – parsed into a `#headers` record
- The HTTP request – parsed into a `#http_request` record
- `clidata` – data which is `POST`ed by the client
- `querydata` – this is the remainder of the URL following the first occurrence of a '?' character, if any.
- `docroot` – the absolute path to the docroot of the virtual server that is processing the request.

```
-record(arg, {
          clisock,        % the socket leading to the peer client
          client_ip_port, % {ClientIp, ClientPort} tuple
          headers,        % headers
          req,            % request (possibly rewritten)
          orig_req,       % original request
          clidata,        % The client data (as a binary in POST requests)
          server_path,    % The normalized server path
                          % (pre-querystring part of URI)
          querydata,      % For URIs of the form ...?querydata
                          %  equiv of cgi QUERY_STRING
          appmoddata,     % (deprecated - use pathinfo instead) the remainder
                          % of the path leading up to the query
          docroot,        % Physical base location of data for this request
          docroot_mount,  % virtual directory e.g /myapp/ that the docroot
                          %  refers to.
          fullpath,       % full deep 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, % (deprecated - use prepath instead) path in front
                          %  of: <appmod><appmoddata>
          prepath,        % Path prior to 'dynamic' segment of URI.
                          %  ie http://some.host/<prepath>/<script-point>/d/e
                          % where <script-point> is an appmod mount point,
                          % or .yaws,.php,.cgi,.fcgi etc script file.
          pathinfo,       % Set to '/d/e' when calling c.yaws for the request
                          % http://some.host/a/b/c.yaws/d/e
                          %  equiv of cgi PATH_INFO
          appmod_name     % name of the appmod handling a request,
                          % or undefined if not applicable
         }).

-record(http_request, {method,
                       path,
                       version}).


-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,
          location,
          content_length,
          content_type,
          content_encoding,
          authorization,
          transfer_encoding,
          x_forwarded_for,
          other = []   % misc other headers
         }).
```

There are a number of *advanced* fields in the `#arg` record such as `appmod` and `opaque` that will be discussed in later chapters.

Now, we show some code which displays the content of the Arg #arg record. The code is available in yaws/www/arg.yaws and after a local_install a request to http://localhost:8000/arg.yaws will run the code.

```
<html>

<h2> The Arg </h2>

<p>This page displays the Arg #argument structure
supplied to the out/1 function.

<erl>


out(A) ->
    Req = A#arg.req,
    H = yaws_api:reformat_header(A#arg.headers),
    {ehtml,
     [{h4,[], "The headers passed to us were:"},
      {hr},
      {ol, [],lists:map(fun(S) -> {li,[], {p,[],S}} end,H)},

      {h4, [], "The request"},
      {ul,[],
       [{li,[], f("method: ~s",  [Req#http_request.method])},
        {li,[], f("path: ~p",    [Req#http_request.path])},
        {li,[], f("version: ~p", [Req#http_request.version])}]},

      {hr},
      {h4, [], "Other items"},
      {ul,[],
       [{li,[], f("clisock from: ~p", [inet:peername(A#arg.clisock)])},
        {li,[], f("docroot: ~s",      [A#arg.docroot])},
        {li,[], f("fullpath: ~s",     [A#arg.fullpath])}]},
      {hr},
      {h4, [], "Parsed query data"},
      {pre,[], f("~p", [yaws_api:parse_query(A)])},
      {hr},
      {h4,[], "Parsed POST data "},
      {pre,[],  f("~p", [yaws_api:parse_post(A)])}]}.

</erl>

</html>
```

The code utilizes four functions from the yaws_api module. The yaws_api module is a general purpose www API module that contains various functions that are handy while developing YAWS code. We will see many more of those functions during the examples in the following chapters.

The functions used are:

- `yaws_api:f/2` — alias for `io_lib:format/2`. The `f/2` function is automatically `-included` in all YAWS code.
- `yaws_api:reformat_header/1` — This function takes the #headers record and unparses it, that is reproduces regular text.
- `yaws_api:parse_query/1` — The topic of the next section.
- `yaws_api:parse_post/1` — Ditto.

4.3 POSTs

4.3.1 Queries

The user can supply data to the server in many ways. The most common is to give the data in the actual URL. If we invoke:

`GET http://localhost:8000/arg.yaws?kalle=duck&goofy=unknown`

we pass two parameters to the `arg.yaws` page. That data is URL-encoded by the browser and the server can retrieve the data by looking at the remainder of the URL following the '?' character. If we invoke the `arg.yaws` page with the above mentioned URL we get as the result of `yaws_api:parse_query/1`:

$kalle = duck$

$goofy = unknown$

In ERLANG terminology, the call `yaws_api:parse_query(Arg)` returns the list:

```
[{"kalle", "duck"}, {"goofy", "unknown"}]
```

Both the key and the value are strings. Hence, a web page can contain URLs with a query and thus pass data to the web server. This scheme works with any kind of requests. It is the easiest way to pass data to the Web server since no form is required in the web page.

4.3.2 Forms

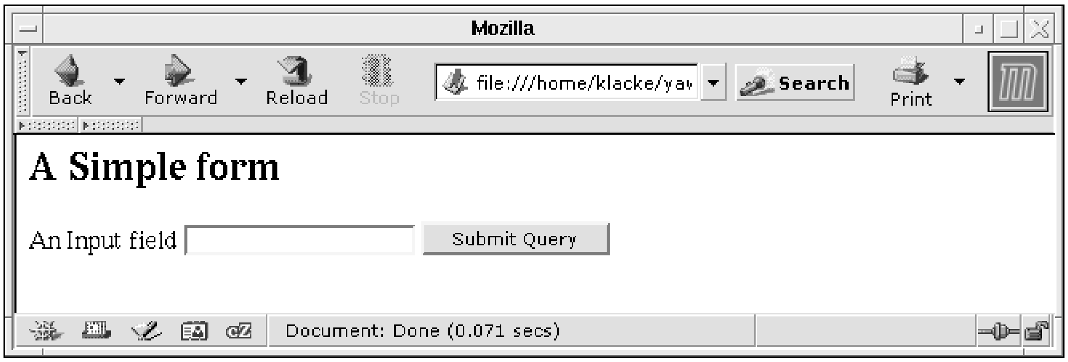
In order to `POST` data a form is required. Say that we have a page called `form.yaws` that contain the following code:

```
<html>
<form action="/post_form.yaws"
      method="post"

<p> A Input field
```

```
<input name="xyz" type="text">
<input type="submit">
</form>
</html>
```

This will produce a page with a simple input field and a submit button.



If we enter something—say, “Hello there”—in the input field and click the submit button the client will request the page indicated in the “action” attribute, namely `post_form.yaws`.

If that YAWS page has the following code:

```
out(A) ->
   L = yaws_api:parse_post(A),
   {html, f("~p", [L])}
```

The user will see the output

```
[{"xyz", "Hello there"}]
```

The differences between using the query part of the URL and a form are the following:

- Using the query arg works with any kind of requests. We parse the query argument with the function `yaws_api:parse_query(Arg)`
- If we use a form and `POST` the user data the client will transmit the user data in the body of the request. That is, the client sends a request to get the page using the `POST` method and it then attaches the user data—encoded—into the body of the request.

  A `POST` request can have a query part in its URL as well as user data in the body.

4.4 POSTing files

It is possible to upload files from the client to the server by means of `POST`. We indicate this in the form by telling the browser that we want a different encoding. Here is an example form that does this:

```
out(A) ->
    Form =
        {form, [{enctype, "multipart/form-data"},
                {method, post},
                {action, "file_upload_form.yaws"}],
                [{input, [{type, submit}, {value, "Upload"}]},
                 {input, [{type,file}, {width, "50"}, {name, foo}]}]},
    {ehtml, {html,[], [{h2,[], "A simple file upload page"},
                      Form]}}.
```

As shown in the figure, the page delivers the entire HTML page with enclosing `html` markers.

The user gets an option to browse the local host for a file or the user can explicitly fill in the file name in the input field. The file browsing part is automatically taken care of by the browser.

The action field in the form states that the client shall POST to a page called `file_upload_form.yaws`. This page will get the contents of the file in the body of the `POST` message. To read it, we use the `yaws_multipart` module, which provides the following capabilities:

1. It reads all parameters — files uploaded and other simple parameters.
2. It takes a few options to help file uploads. Specifically:
   (a) `{max_file_size, MaxBytes}`: if the file size in bytes exceeds `MaxBytes`, return an error
   (b) `no_temp_file`: read the uploaded file into memory without any temp files
   (c) `{temp_file,FullFilePath}`: specify `FullFilePath` for the temp file; if not given, a unique file name is generated
   (d) `{temp_dir, TempDir}`: specify `TempDir` as the directory to store the uploaded temp file; if this option is not provided, then by default an OS-specific temp directory such as `/tmp` is used
   (e) `list`: return file data in list form; this is the default
   (f) `binary`: return file data in binary form
   (g) `return_error_file_path`: if an error occurs writing to a file or if `max_file_size` is exceeded, return the file pathname as part of the error (more on this below).

Note that the `list` and `binary` options affect only file data, not filenames, headers, or other parameters associated with each file. These are always returned as strings.

Call `yaws_multipart:read_multipart_form` from your `out/1` function and it returns a tuple with the first element set to one of these three atoms:

- `get_more`: more data needs to be read; return this tuple directly to YAWS from your `out/1` function and it will call your `out/1` function again when it has read more `POST` data, at which point you must call `read_multipart_form` again
- `done`: multipart form reading is complete; a `dict` full of parameters is returned
- `error`: an error occurred

The `dict` returned with `done` allows you to query it for parameters by name. For file upload parameters, it returns one of the following lists:

```
[{filename, "name of the uploaded file as entered on the form"},
 {value, Contents_of_the_file_all_in_memory} | _T]
```

or:

```
[{filename, "name of the uploaded file as entered on the form"},
 {temp_file, "full pathname of the temp file"} | _T]
```

Some multipart/form messages also headers such as `Content-Type` and `Content-Transfer-Encoding` for different subparts of the message. If these headers are present in any subpart of a multipart/form message, they're also included in that subpart's parameter list, like this:

```
[{filename, "name of the uploaded file as entered on the form"},
 {value, Contents_of_the_file_all_in_memory},
 {content_type, "image/png"} | _T]
```

Note that for the temporary file case, it's your responsibility to delete the file when you're done with it. To ensure that you can do this even when errors occur, include `return_error_file_path` in the options you pass to `yaws_multipart:read_multipart_form/2`. Should an error occur, the returned error tuple will be of this form:

```
{error, {Reason, ParamName, FileName}}
```

If you supplied the `{temp_file, FullFilePath}` option, `FileName` in the error tuple is the same as `FullFilePath`, otherwise it's the name of a temporary file YAWS generated.

If the `return_error_file_path` option is not included, the returned error tuple will instead be:

```
{error, {Reason, ParamName}}
```

Here's an example of calling `yaws_multipart:read_multipart_form/2`:

```
-module(my_yaws_controller).
-export([out/1]).

out(Arg) ->
    Options = [no_temp_file],
    case yaws_multipart:read_multipart_form(Arg, Options) of
        {done, Params} ->
            io:format("Params : ~p~n", [Params]),
            {ok, [{filename, FileName},{value,FileContent}|_]} =
                dict:find("my_file", Params),
            AnotherParam = dict:find("another_param", Params);
        %% do something with FileName, FileContent and AnotherParam
        {error, Reason} ->
            io:format("Error reading multipart form: ~p~n", [Reason]);
        Other -> Other
    end.
```

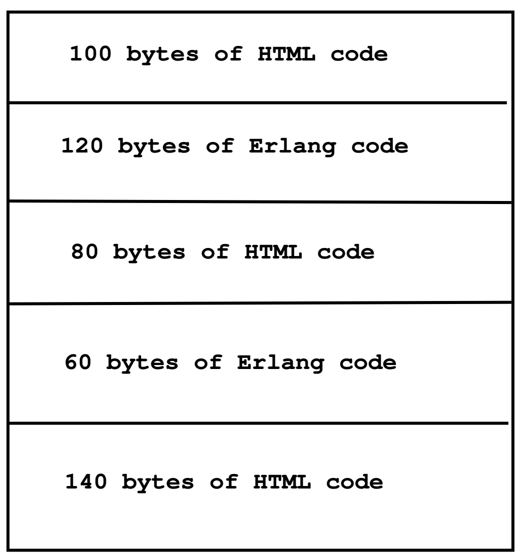
Here, `my_yaws_controller` is a user-defined module compiled as usual with `erlc` with the resulting `.beam` file placed in the YAWS load path. The module is then registered with YAWS as an *appmod* to allow it to receive and process requests—see section **??** for more details.

Chapter 5

Mode of operation

5.1 On-the-fly compilation

When the client requests a YAWS page, YAWS will look in its caches (there is one cache per virtual server) to see if it finds the requested page in the cache. If YAWS doesn't find the page in the cache, it will compile the page. This only happens the first time a page is requested. Say that the page is 400 bytes big and has the following layout:



The YAWS server will then parse the file and produce a structure which makes it possible to readily deliver the page without parsing the next time the same page is requested.

When shipping the page it will

1. Ship the first 100 bytes from the file
2. Evaluate the first ERLANG chunk in the file and ship the output from the `out/1` function in that chunk. It will also jump ahead in the file and skip 120 bytes.
3. Ship 80 bytes of HTML code

4. Again evaluate an ERLANG chunk, this time the second and jump ahead 60 bytes in the file.
5. And finally ship 140 bytes of HTML code to the client

YAWS writes the source output of the compilation into a directory `/tmp/yaws/$UID`. The beam files are never written to a file. Sometimes it can be useful to look at the generated source code files, for example if the YAWS/ERLANG code contains a compilation error which is hard to understand.

5.2 Evaluating the Yaws Code

All client requests will execute in their own ERLANG process. For each group of virtual hosts on the same IP:PORT pair one ERLANG process listens for incoming requests.

This process spawns acceptor processes for each incoming request. Each acceptor process reads and parses all the HTTP headers from the client. It then looks at the `Host:` header to figure out which virtual server to use, i.e. which docroot to use for this particular request. If the `Host:` header doesn't match any server from *yaws.conf* with that IP:PORT pair, the first one from *yaws.conf* is chosen.

By default YAWS will not ship any data at all to the client while evaluating a YAWS page. The headers as well as the generated content are accumulated and not shipped to the client until the entire page has been processed.

Chapter 6

SSL

Secure Socket Layer (SSL) is a protocol used on the Web for delivering encrypted pages to the WWW client. SSL is widely deployed on the Internet and virtually all bank transactions as well as all online shopping today is done with SSL encryption. There are many good sources on the net that describe SSL in detail, so we will not try to do that here. See for example `http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/`, which describes how to manage certificates and keys.

In order to run an SSL server we must have a certificate. Either we can create a so-called self-signed certificate ourselves or buy a certificate from one of the many CA's (Certificate Authorities) on the net. YAWS uses the SSL application provided by Erlang/OTP and depends on features supported by it. So, from an Erlang/OTP release to another, some SSL options may change, others may appear or disappear. Check the documentation of your Erlang/OTP release for details.

To setup a YAWS server with SSL we could have a *yaws.conf* file that looks like:

```
logdir = /var/log/yaws

<server www.funky.org>
    port = 443
    listen = 192.168.128.32
    docroot = /var/yaws/www.funky.org
    <ssl>
        keyfile = /etc/funky.key
        certfile = /etc/funky.cert
        password = gazonk
    </ssl>
</server>
```

This is the easiest possible SSL configuration. The configuration refers to a certificate file and a key file. The certificate file must contain the name "www.funky.org" as it "Common Name".

The keyfile is the private key file and it is encrypted using the password "gazonk".

6.1 Server Name Indication

The SNI support was introduced in the SSL application in Erlang/OTP 18.0. It is an extension to the TLS protocol (RFC 4366), which allows the client to include the requested hostname in the first message of its SSL handshake.

With SNI, you can have many virtual server sharing the same IP address and port, and each one can have its own unique certificate (and the rest of the configuration). If SNI is disabled or not supported, all virtual servers on the same IP/Port must share the same SSL configuration. In this situation, it is complicated for YAWS to present a valid SSL certificate to clients. Possible solutions are listed here: `http://wiki.cacert.org/VhostTaskForce`.

By default, SNI was originally disabled in YAWS to be backward compatible with old Erlang/OTP releases; it remains disabled by default to remain compatible with older YAWS releases. But it can be enabled and fine-tuned for each SSL server. Here is a basic example:

```
logdir = /var/log/yaws
sni = enable

<server www.example.com>
    port = 443
    listen = 192.168.128.32
    docroot = /var/yaws/www.example.com
    <ssl>
        keyfile = /etc/ssl/www.example.com.key
        certfile = /etc/ssl/www.example.com.cert
    </ssl>
</server>

<server www.example.org>
    port = 443
    listen = 192.168.128.32
    docroot = /var/yaws/www.example.org
    <ssl>
        keyfile = /etc/ssl/www.example.org.key
        certfile = /etc/ssl/www.example.org.cert
    </ssl>
</server>
```

Depending on the SNI hostname provided by the client, the first or the second virtual host will be chosen, and the corresponding SSL certificate will be presented to the client. In this example, non-SNI clients are still supported. For such clients, the SSL certificate of the first virtual server will be presented and the HTTP Host header will then be used to find the correct virtual server. Otherwise, it is possible to refuse non-SNI clients, globally or per server.

Chapter 7

Applications

YAWS is well suited for Web applications. In this chapter we will describe a number of application templates. Code and strategies that can be used to build Web applications.

There are several ways of starting applications from YAWS.

- The first and most easy variant is to specify the `-r Module` flag to the YAWS startup script. This will `apply(Module,start,[])`
- We can also specify `runmods` in the *yaws.conf* file. It is possible to have several modules specified if want the same YAWS server to run several different applications.

```
runmod = myapp
runmod = app_number2
```

- It is also possible to do it the other way around, let the main application start YAWS. We call this embedded mode, which we will discuss in chapter **??**.

7.1 Login scenarios

Many Web applications require the user to login. Once the user has logged in the server sets a cookie and then the user will be identified by help of the cookie in subsequent requests.

7.1.1 The session server

The cookie is passed in the headers and is available to the YAWS programmer in the `Arg #arg` record. The YAWS session server can help us to maintain a state for a user while the user is logged in to the application. The session server has the following 5 API functions to aid us:

1. `yaws_api:new_cookie_session(Opaque)` — This function initiates a new cookie-based session. The `Opaque` data is typically some application-specific structure which makes it possible for the application to read a user state, or it can be the actual user state itself.

2. `yaws_api:cookieval_to_opaque(Cookie)` — This function maps a cookie to a session.

3. `yaws_api:replace_cookie_session(Cookie, NewOpaque)` — Replace the opaque user state in the session server with `NewOpaque`.

4. `yaws_api:delete_cookie_session(Cookie)` — This function should typically be called when the user logs out or when our web application decides to automatically logout the user.

In addition, `yaws_session_server` allows you to create your own cookies if you prefer not to use its built-in cookie generator. To supply your own session cookie generator, set the global configuration variable `ysession_cookiegen` to the name of a module that provides a `new_cookie/0` function that returns a cookie as a list. If you specify a cookie generator module in your configuration, please be sure that its `new_cookie/0` function returns a unique cookie each time it's called.

All cookie-based applications are different but they have some things in common. In the following examples we assume the existence of a function `myapp:auth(UserName, Passwd)` and it returns `ok` or `{error, Reason}`.

Let's assume the following record:

```
-record(session, {user,
                  passwd,
                  udata = []}).
```

The following function is a good template function to check the cookie.

```
get_cookie_val(CookieName, Arg) ->
    H = Arg#arg.headers,
    yaws_api:find_cookie_val(CookieName, H#headers.cookie).



check_cookie(A, CookieName) ->
    case get_cookie_val(CookieName, A) of
        []  ->
            {error, "not logged in"};
        Cookie ->
            yaws_api:cookieval_to_opaque(Cookie)
    end.
```

We need to check all requests and make sure the session_server has our cookie registered as an active session. Also, if a request comes in without a working cookie we want to present a login page instead of the page the user requested. Another quirky issue is that the pages necessary for display of the login page must be shipped without checking the cookie. The next sections explain how these needs can be met.

7.1.2 Arg rewrite

In this section we describe a feature whereby the user is allowed to rewrite the `Arg` at an early stage in the YAWS server. We do that by specifying an `arg_rewrite_mod` in the *yaws.conf* file.

```
arg_rewrite_mod = myapp
```

Then in the `myapp` module we have:

```
arg_rewrite(Arg) ->
    OurCookieName = "myapp_sid"
    case check_cookie(A, OurCookieName) of
        {error, _} ->
            do_rewrite(Arg);
        {ok, _Session} ->
            %return Arg untouched
            Arg
    end.

%% these pages must be shippable without a good cookie
login_pages() ->
    ["/banner.gif", "/login.yaws", "/post_login.yaws"].

do_rewrite(Arg) ->
    Req = Arg#arg.req,
    {abs_path, Path} = Req#http_request.path,
    case lists:member(Path, login_pages()) of
        true ->
            Arg;
        false ->
            Arg#arg{req = Req#http_request{path = {abs_path, "/login.yaws"}},
                    state = Path}
    end.
```

Our arg rewrite function lets all `Args` go through untouched that either have a good cookie or belong to a set of predefined pages that are acceptable to get without being logged in. If we decide that the user must log in, we change the path of the request, thereby making the YAWS server ship a login page instead of the page the user requested. We also set the original path in the `Arg` state argument so that the login page can redirect the user to the original page once the login procedure is finished.

Within an arg rewrite function, examining and modifying HTTP headers can be achieved using the following functions from the `yaws_api` module:

- `set_header/2`, `set_header/3`
- `get_header/2`, `get_header/3`

- merge_header/2, merge_header/3
- delete_header/2

All functions operate on instances of the #headers{} record type defined in the yaws_api.hrl include file.

Should an arg rewrite function need to return a response for a particular request, it can do so by returning an #arg record instance with its state field set to an instance of a #rewrite_response record. For example, below is an arg rewrite function that allows only GET and POST requests, returning a 405 *Method Not Allowed* for requests with any other HTTP method:

```
arg_rewrite(Arg) ->
    Allowed = ['GET', 'POST'],
    Req = yaws_api:arg_req(Arg),
    Method = yaws_api:http_request_method(Req),
    case lists:member(Method, Allowed) of
        true ->
            Arg;
        false ->
            AllowedStrs = string:join([atom_to_list(M) || M <- Allowed], ","),
            Arg#arg{state=
                        #rewrite_response{
                           status=405,
                           headers=[{header, {"Allow", AllowedStrs}},
                                    {header, {connection, "close"}}]}}
    end.
```

7.1.3 Authenticating

Now we're approaching the login.yaws page, the page that displays the login prompt to the user. The login page consists of two parts: one part that displays the login data as a form, and one form processing page that reads the data the user entered in the login fields and performs the actual authentication.

The login page performs a tiny well-known Web trick where it passes the original URL request in a hidden field in the login page and thereby passing that information to the form processing page.

The page login.yaws:

```
<erl>

out(A) ->
    {ehtml,
     {html,[],
      [{h2, [], "Login page"},
       {hr},
       {form, [{action,"/login_post.yaws"},
               {method,post}],
```

```
        [{p,[], "Username"}, {input, [{type,text},{name,uname}]},
         {p,[],"Password"},  {input, [{type,password},{name,passwd}]},
         {input, [{type,submit},{value,"Login"}]},
         {input, [{type,hidden},{name,url},
                  {value, A#arg.state}]}]}]}]}}.

</erl>
```

The form processing page which gets the `POST` data from the code above:

```
<erl>

-include("myapp.hrl").
%% we have the session record there
%% we must set the include_path in the yaws.conf file
%% in order for the compiler to find that file

kv(K,L) ->
    {value, {K, V}} = lists:keysearch(K,1,L),
    V.

out(A) ->
    L = yaws_api:parse_post(A),
    User = kv(user, L),
    Pwd =  kv(passwd, L),
    case myapp:auth(User, Pwd) of
        ok ->
            S = #session{user = User,
                         passwd = Pwd,
                         udata = []},
            %% Now register the session to the session server
            Cookie = yaws_api:new_cookie_session(S),
            [{redirect_local, kv(url, L)},
              yaws_api:set_cookie("myapp_sid",Cookie,[])]
        Err ->
            {ehtml,
             {html, [],
              {p, [], f("Bad login: ~p",[Err])}}}
    end.

</erl>
```

The function returns a list of two new (not previously discussed) return values: instead of returning HTML output as in `{html, Str}` or `{ehtml,Term}` we return a list of two new values. (There are many different possible return values from the `out/1` function and they will all be described later.) The two new values are:

1. The tuple `{redirect_local, Path}` makes the YAWS web server return a 302 redirect to the specified `Path`. Optionally a different status code can be supplied which will be used in place of 302, e.g. `{redirect_local, Path, 307}`.
2. `yaws_api:set_cookie("myapp_sid",Cookie,[])` generates a `Set-Cookie` header.

Now if we put all this together we have a full-blown cookie-based login system. The last thing we did in the form processing code was to register the session with the session server thereby letting any future requests go straight through the `Arg` rewriter.

This way both YAWS pages as well as all or some static content is protected by the cookie login code.

7.1.4 Database driven applications

We can use code similar to the code in the previous section to associate a user session to entries in a database. Mnesia fits perfectly together with YAWS and keeping user persistent state in Mnesia is both easy and convenient.

Once the user has logged in we can typically use the user name as key into the database. We can mix `ram_tables` and `disc_tables` to our liking. The Mnesia database must be initialized via `create_table/2` before it can be used. This is typically done while installing the web application on a machine.

Another option is to let the application check that Mnesia is initialized whenever the application starts.

If we don't want or need to use Mnesia, it's of course possible to use a simple `dets` file or a text file as well.

7.2 Appmods

Appmods is mechanism to invoke different applications based upon the URL. A URL—as presented to the web server in a request—has a path part and a query part.

It is possible to install several appmods in the *yaws.conf* file as shown below:

```
appmods = foo myapp
```

Now, if the user requests a URL where any component in the directory path is an appmod, the parsing of the URL will terminate there and instead of reading the actual file from the disk, YAWS will invoke the appmod with the remainder of the path inserted into `Arg#arg.appmoddata`.

Say the user requests the URL `http://www.funky.org/myapp/xx/bar.html`. YAWS will not ship the file `bar.html` to the client, instead it will invoke `myapp:out(Arg)` with `Arg#arg.appmoddata` set to the string `xx/bar.html`. Any optional query data—that is, data that follows the first '?' character in the URL—is removed from the path and passed as `Arg#arg.querydata`.

Appmods can be used to run applications on a server. All requests to the server that has an appmod in the URL will be handled by that application. If the application decides that it want to ship a page from the disk to the client, it can return the tuple `{page, Path}`. This return value will make YAWS read the page from the disk, possibly add the page to its cache of commonly accessed pages and ship it back to the client.

The `{page, Path}` return value is equivalent to a redirect, but it removes an extra round trip, and is thus faster.

Appmods can also be used to fake entire directory hierarchies that don't exist on disk.

7.3 The opaque data

Sometimes an application needs application-specific data such as the location of its data files. There exists a mechanism to pass application-specific configuration data from the YAWS server to the application.

When configuring a server we have an opaque field in the configuration file that can be used for this purpose. Say we have the following fields in the config file:

```
<server foo>
    listen = 192.168.128.44
    <opaque>
        foo = bar
        somefile = /var/myapp/db
        myname = example
    </opaque>
</server>
```

This will create a normal server that listens to the specified IP address. An application has access to the opaque data that was specified in that particular server through `Arg#arg.opaque`.

If we have the opaque data specified above, the `Arg` opaque field will have the value:

```
[{foo, "bar"},
 {somefile, "/var/myapp/db"},
 {myname, "example"}
]
```

7.4 Customizations

When actually deploying an application at a live site, some of the standard YAWS behaviors are not acceptable. Many sites want to customize the web server behavior when a client requests a page that doesn't exist on the web server. The standard YAWS behavior is to reply with status code 404 and a message explaining that the page doesn't exist.

Similarly, when YAWS code crashes, the reason for the crash is displayed in the Web browser. This is very convenient while developing a site but not acceptable in production.

7.4.1 404 File not found

We can install a special handler for 404 messages. We do that by specifying a `errormod_404` in the *yaws.conf* file.

If we have:

```
<server foo>
  ..
  ..
  ..
  errormod_404 = myapp

</server>
```

When YAWS gets a request for a file that doesn't exist, it invokes the `errormod_404` module to generate both the status code as well as the content of the message.

If `Module` is specified as the `errormod_404` module, YAWS will invoke `Module:out404(Arg, GC, SC)`, passing the arguments as described below:

- `Arg` is an `#arg{}` record
- `GC` is a `#gconf{}` record (defined in `yaws.hrl`)
- `SC` is a `#sconf{}` record (defined in `yaws.hrl`)

The function can and must do the same things that a normal `out/1` function does.

7.4.2 Crash messages

We use a similar technique for generating the crash messages: we install a module in the *yaws.conf* file and let that module generate the crash message. We have:

```
errormod_crash = Module
```

The default is to display the entire formatted crash message in the browser. This is good for debugging but not good for production.

If `Module` is specified as the `errormod_crash` module, the function `Module:crashmsg(Arg, SC, Str)` will be called. The `Str` argument is the real crash message formatted as a string.

7.5 Stream content

If the `out/1` function returns the tuple `{content, MimeType, Content}` YAWS will ship that data to the Client. This way we can deliver dynamically generated content to the client which is of a different MIME type than "text/html".

If the generated response is very large and it not possible or practical to generate the whole thing, we can return the value:

```
{streamcontent, MimeType, FirstChunk}
```

which delivers data back to the client using HTTP chunked transfer (see RFC 2616 section 3.6.1) and then from a different ERLANG process deliver the remaining chunks by using the functions described below:

1. `yaws_api:stream_chunk_deliver(YawsPid, Data)` where the `YawsPid` is the process id of the YAWS worker process. That pid is available in `Arg#arg.pid`.
2. `stream_chunk_end(YawsPid)` which must be called to indicate the end of the stream.

A streaming alternative is also available for applications that need a more direct way to deliver data to clients, such as those dealing with data too large to buffer in memory but not wishing to use chunked transfer, or applications that use long-polling (Comet) techniques that require them to hold client connections open for extended periods. For these situations we can return the value:

```
{streamcontent_from_pid, MimeType, Pid}
```

to tell YAWS we wish to deliver data of MIME type `MimeType` to the client from process `Pid`. In this case, YAWS will prepare the socket for delivery from `Pid` and then send one of the following messages to `Pid`:

- `{ok, YawsPid}` tells `Pid` that it is now OK to proceed with sending data back to the client using the socket. The socket is accessible as `Arg#arg.clisock`.
- `{discard, YawsPid}` informs `Pid` that it should not attempt to use the socket, typically because the requested HTTP method requires no response body.

We call one of the following functions to send data:

- `yaws_api:stream_process_deliver(Socket, IoList)` sends data `IoList` using socket `Socket` without chunking the data. To ensure chunking is not in effect, return

  ```
  {header, {transfer_encoding, erase}}
  ```

  in a list with the `streamcontent_from_pid` directive (which must always be last in such a list).
- `yaws_api:stream_process_deliver_chunk(Socket, IoList)` sends data `IoList` using socket `Socket` but converts the data into chunked transfer form before sending it.

Pids using chunked transfer must indicate the end of their transfer by calling the following function:

- `yaws_api:stream_process_deliver_final_chunk(Socket, IoList)`

which delivers a special HTTP chunk to mark the end of the data transfer to the client.

Finally, `Pid` must always call `yaws_api:stream_process_end(Socket, YawsPid)` when it finishes sending data or when it receives the `{discard, YawsPid}` message from YAWS — this is required to inform YAWS that `Pid` has finished with the socket and will not use it directly anymore. If the application has to close the socket while it's in control of it, though, it must pass the atom `closed` as the first argument to `yaws_api:stream_process_end` in place of the socket to inform YAWS that the socket has been closed and it should no longer attempt to use it.

Applications that use the `streamcontent_from_pid` directive that also want to avoid chunked transfer encoding for their streams should be sure to include a setting for the `Content-Length` header in their `out/1` return value. YAWS automatically sets the `Transfer-Encoding` header to `chunked` if it does not detect a `Content-Length` header.

7.6 All out/1 Return Values

- `{html, DeepList}` This assumes that `DeepList` is formatted HTML code. The code will be inserted in the page.
- `{ehtml, Term}` This will transform the ERLANG term `Term` into a stream of HTML content.
- `{exhtml, Term}` This will transform the ERLANG term `Term` into a stream of strict XHTML 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 using HTTP chunked transfer 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 the 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 the 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, Pid}` This 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 *c̈lose¨*, 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, What}` - 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, What}` - Appends a *Set-Cookie:* header to the list of previously set *Set-Cookie* headers.
  – `{content_range, What}` - Sets the *Content-Range:* header.
  – `{content_type, What}` - Sets the *Content-Type:* header.
  – `{content_encoding, What}` - Sets the *Content-Encoding:* header. If this header is defined, no deflate is performed by YAWS , allowing you to compress data yourself if you wish to do so.
  – `{content_length, What}` - 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.
  – `{vary, What}` - Sets the *Vary:* header.
  – `{accept_ranges, What}` - Sets the *Accept-Ranges:* header.

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

  ```
  {header, {"My-X-Header", "gadong"}} or {header, "My-X-Header: gadong"}
  ```

- `{header, {H, erase}}` A specific case of the previous directive; use this to remove a specific header from a response. For example, streaming applications and applications using server-sent events (see `http://www.w3.org/TR/eventsource/`) should use `{header, {transfer_encoding, erase}}` to turn off chunked encoding for their responses.

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

- `{status, Code}` Sets the response HTTP status code to `Code`.

- `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 HTTP `Location` header. Set the status code to 302.
- `{redirect, Url, Status}` Same as redirect above with the additional option of supplying the status code. The default for a redirect is 302 but 301, 303 and 307 are also valid redirect status codes.
- `{redirect_local, Path}` Does a redirect to the same `Scheme://Host:Port/Path` in which we are currently executing. `Path` can be either be the path directly (equivalent to `abs_path`), or one of `{{abs_path, Path}` or `{{rel_path, RelativePath}}`
- `{redirect_local, Path, Status}` Same as `redirect_local` above with the additional option of supplying the status code. The default for a redirect is 302 but 301, 303 and 307 are also valid redirect status codes.
- `{get_more, Cont, State}` When we are receiving large `POST`s we can return this value and be invoked again when more data arrives.
- `{page, Page}` Make YAWS returns a different local page than the one being requested. `Page` is a Request-URI, so it must be url-encoded and can contain a query-string.
- `{page, {Options, Page}}` Like the above, but supplying an additional deep list of options. Supported options types are:
  - `{status, C}` - Set the HTTP response status code `C` for page `Page`.
  - `{header, H}` - Accumulate the HTTP header `H` for page `Page`.
  - `{disable_cache, Bool}` - if set to `true`, disable the cache of `Page` for this call.
- `{websocket, CallbackModule, Options}` Tell YAWS to use `CallbackModule` as a WebSocket Protocol handler for traffic on the client socket. See chapter **??** for more details.
- `{ssi, File, Delimiter, Bindings}` Server-side include `File` and macro expansion in `File`. Each occurrence of a string, say "xyz", inside `File` that's within a `Delimiter` pair is replaced with the corresponding value in `Bindings`. For 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 within a deep ehtml structure.

  The special directive `strip_undefined` can be specified in the `Bindings` list, just as it can for the `{bindings, ....}` directive, but it's ignored because treating undefined variables as empty is the default for `ssi` bindings.

- {bindings, [{Key1, Value2}, {Key2, Value2} .....]} Establish variable bindings that can be used in the content returned from out/1. 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) (which calls error if no such binding exists) or yaws_api:binding_find(Key) (which returns undefined if no such binding exists) can be used to extract Value, and yaws_api:binding_exists(Key) can be used to check for the existence of a binding.

  If content to be returned happens to contains text that looks like a binding, e.g. %%SomeText%%, but no key SomeText is supplied, then by default the original text is left as is. If you prefer that anything parsed as a binding gets stripped out of content whenever the bindings directive does not specify its key, include the special directive strip_undefined in the bindings list:

  ```
  {bindings, [{Key1, Value1}, strip_undefined]}
  ```

  The same result of stripping undefined bindings out of returned out/1 content can be achieved by setting strip_undefined_bindings = true in the server configuration.

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

- #arg{} Return an instance of an #arg{} record. This can be useful when used as part of a [ListOfValues] return value, so that any subsequent elements in the return list that require an #arg{} get the returned instance rather than the original. For example, an out/1 function might set the state field of an #arg{}, then return both it and {yssi, YawsFile} in a list, in which case YAWS will pass the returned #arg{}, rather than the original instance, to the yaws file out/1 function.

- [ListOfValues] It is possible to return a list of the above defined return values. Any occurrence of the atoms 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.

Chapter 8

Debugging and Development

YAWS has excellent debugging capabilities. First and foremost we have the ability to run the web server in interactive mode by means of the command line switch `-i`, which gives us a regular ERLANG command line prompt we can use to compile helper code or reload helper code. Furthermore all error messages are displayed there. If a `.yaws` page producees any regular ERLANG I/O, that output will be displayed at the ERLANG prompt, assuming we are running in interactive mode.

If we give the command line switch `-d` we get some additional error messages. Also YAWS does some additional checking of user supplied data such as headers.

8.1 Logs

YAWS produces various logs. All log files are written into the YAWS logdir directory. This directory is specified in the config file.

We have the following log files:

- The access log. Access logging is turned on or off per server in the *yaws.conf* file. If access_log is turned on for a server, YAWS will produce a log in Common Access Log Format called *HostName:PortNumber.access*
- The auth log. Auth logging is turned on or off per server in the *yaws.conf* file. If auth_log is turned on for a server, YAWS will produce a log called *HostName:PortNumber.auth* which contains all HTTP auth-related messages.
- *report.log* This file contains all error and crash messages for all virtual servers in the same file.
- Trace files. The two command line flags `-t` and `-T` tells YAWS to trace all traffic or just all HTTP messages and write them to files.

Chapter 9

External scripts via CGI

YAWS can also interface to external programs generating dynamic content via the Common Gateway Interface (CGI). This has to be explicitly enabled for a virtual host by listing `cgi` in the `allowed_scripts` line in the configuration file. Any request for a page ending in `.cgi` (or `.CGI`) will then result in trying to execute the corresponding file.

If you have a PHP executable compiled for using CGI in the `PATH` of the YAWS server, you can enable PHP support by adding `php` to `allowed_scripts`. Requests for pages ending in `.php` will then result in YAWS executing `php` (configurable via `php_handler`) and passing the name of the corresponding file to it via the appropriate environment variable.

These ways of calling CGI scripts are also available to `.yaws` scripts and appmods via the functions `yaws_api:call_cgi/2` and `yaws_api:call_cgi/3`. This makes it possible to write wrappers for CGI programs, irrespective of the value of `allowed_scripts`.

The author of this YAWS feature uses it for self-written CGI programs as well as for using a standard CGI package. You should not be surprised however, should some scripts not work as expected due to an incomplete or incorrect implementation of certain CGI meta-variables. The author of this feature is interested in hearing about your experiences with it. He can be contacted at `carsten@codimi.de`.

Chapter 10

FastCGI

YAWS supports the responder role and the authorizer role of the FastCGI protocol. See `www.fastcgi.com` for details on the FastCGI protocol.

The benefits of using FastCGI include:

1. Unlike CGI, it is not necessary to spawn a new process for every request; the application server can handle multiple requests in a single process.
2. The fact that the application server can run on a different computer benefits scalability and security.
3. The application server can be written in any language for which a FastCGI library is available. Existing applications which have been written for other web servers can be used with YAWS.
4. FastCGI can also be used to implement external authentication servers (in addition to generating dynamic content).

Support for FastCGI was added to YAWS by Bruno Rijsman (`brunorijsman@hotmail.com`).

10.1 The FastCGI Responder Role

The FastCGI responder role allows YAWS to communicate with an application server running on a different (or on the same) computer to generate dynamic content.

The FastCGI protocol (which runs over TCP) is used to send the request information from YAWS to the application server and to send the response information (e.g. the generated dynamic content) from the application server back to YAWS.

FastCGI responders can be invoked in two ways:

1. By including `fcgi` in the `allowed_scripts` line in the configuration file (note that the default value for `allowed_scripts` includes `fcgi`).

   In this case a request for any resource with the `.fcgi` extension will result in a FastCGI call to the application server to dynamically generate the content.

   Note: the YAWS server will only call the application server if a file corresponding to the resource name (i.e. a file with the `.fcgi` extension) exists locally on the YAWS server. The contents of that file are not relevant.

2. By creating an appmod which calls `yaws_api:call_fcgi_responder`. See the `yaws_api(5)` man page for details.

10.2 The FastCGI Authorizer Role

The FastCGI authorizer role allows YAWS to communicate with an authentication server to authenticate requests.

The FastCGI protocol is used to send the request information from YAWS to the authentication server and the authentication response back from the authentication server to YAWS.

If access is allowed, YAWS proceeds to process the request normally.

If access is denied, the authentication server provides the response which is sent back to the client. This is typically a "not authorized" response or a redirect to a login page.

FastCGI authorizers are invoked by creating an appmod which calls `yaws_api:call_fcgi_authorizer`. See the `yaws_api(5)` man page for details.

10.3 The FastCGI Filter Role

FastCGI defines a third role, the filter role, which YAWS does not currently support.

10.4 FastCGI Configuration

The following commands in the *yaws.conf* file control the operation of FastCGI.

If you use FastCGI, you *must* include the `fcgi_app_server` setting in the configuration file to specify the host name (or IP address) and TCP port of the FastCGI application server.

You may include the `fcgi_trace_protocol` setting to enable or disable tracing of FastCGI protocol messages. This is useful for debugging.

You may include the `fcgi_log_app_error` setting to enable or disable logging application errors (any output to stderr and non-zero exit codes).

You may include the `extra_cgi_vars` command to pass additional environment variables to the application.

Chapter 11

Security

YAWS is of course susceptible to intrusions. YAWS has the ability to run under a different user than root, even if we need to listen to privileged port numbers. Running as root is generally a bad idea.

Intrusions can happen basically at all places in YAWS code where the YAWS code calls either the BIF `open_port` or when YAWS code calls `os:cmd/1`. Both `open_port` and `os:cmd/1` invoke the `/bin/sh` interpreter to execute its commands. If the commands are nastily crafted bad things can easily happen.

All data that is passed to these two function must be carefully checked.

Since YAWS is written in ERLANG a large class of cracks are eliminated since it is not possible to perform any buffer overrun cracks on a YAWS server. This is very good.

Another possible point of entry to the system is by providing a URL which takes the client out from the docroot. This should not be possible – and the impossibility relies on the correctness of the URL parsing code in YAWS.

11.1 WWW-Authenticate

YAWS has support for WWW-Authentication. WWW-Authenticate is a standard HTTP scheme for the basic protection of files with a username and password. When a client browser wants a protected file, it must send a `Authenticate: username:password` header in the request. Note that this is plain text. If there is no such header or the username and password is invalid the server will respond with status code 401 and the realm. Browsers will then tell the user that a username and password is needed for "realm", and will resend the request after the user enters the information.

WWW-Authentication is configured in the *yaws.conf* file, in as many *<auth>* directives as you desire:

```
<server foo>
  docroot = /var/yaws/www/

..
..

  <auth>
    realm = secretpage
```

```
    dir   = /protected
    dir   = /anotherdir
    user  = klacke:gazonk
    user  = jonny:xyz
    user  = ronny:12r8uyp09jksfdge4
  </auth>
</server>
```

YAWS will require one of the given username:password pairs for all files in the */protected* and */anotherdir* directories. Note that these directories are specified as a server path, that is, the filesystem path that is actually protected here is */var/yaws/www/protected* .

Chapter 12

Embedded mode

YAWS is a normal OTP application. It is possible to integrate YAWS into another larger application. The YAWS source tree must be integrated into the larger application's build environment. YAWS is then simply started by `application:start()` from the larger application's boot script, or the YAWS components needed for the larger application can be started individually under the application's supervisor(s).

By default YAWS reads its configuration data from a config file, the default is */usr/local/etc/yaws/yaws.conf* . If YAWS is integrated into a larger application, however, that application typically has its configuration data kept at some other centralized place. Sometimes we may not even have a file system to read the configuration from if we run a small embedded system.

YAWS reads its application environment. If the environment key `embedded` is set to `true`, YAWS starts in embedded mode. Once started it must be fed a configuration, and that can be done after YAWS has started by means of the function `yaws_api:setconf/2`.

It is possible to call `setconf/2` several times to force YAWS to reread the configuration.

12.1 Creating Global and Server Configurations

The `yaws_api:setconf/2` function mentioned in the previous section takes two arguments:

- a `#gconf` record instance, specifying global YAWS configuration
- a list of lists of `#sconf` record instances, each specifying configuration for a particular server instance

These record types are specified in `yaws.hrl`, which is not normally intended for inclusion by applications. Instead, YAWS provides the `yaws_api:embedded_start_conf/1,2,3,4` functions that allow embedded mode applications to specify configuration data using property lists (lists of `{key, value}` pairs).

The `yaws_api:embedded_start_conf` functions all return a tuple containing the following four items:

- the atom `ok`.
- a list of lists of `#sconf` record instances. This variable is intended to be passed directly to `yaws_api:setconf/2` as its second argument.

- a #gconf record instance. This variable is intended to be passed directly to yaws_api:setconf/2 as its first argument.
- a list of supervisor child specification for the YAWS components the embedded mode application's configuration specified should be started. This allows embedded mode applications to start YAWS under its own supervisors.

Note that yaws_api:embedded_start_conf does not actually start any servers, but rather it only returns the configuration information and child specifications needed for the embedded mode application to start and configure YAWS itself.

If you have difficulty figuring out how to set up your #gconf or #sconf records for embedded mode, you should first consider getting something running in non-embedded mode using a yaws.conf file. Once you're satisfied with the setup and YAWS is running, execute the following command:

```
yaws --running-config
```

This command will show the YAWS configuration from your yaws.conf file in terms of #gconf and #sconf records, thus showing you how to set up those records for embedded mode.

12.2 Starting Yaws in Embedded Mode

An embedded mode application can start YAWS in one of two ways:

- It can call yaws_api:embedded_start_conf to obtain configuration and YAWS startup information as described in the previous section, start YAWS under its own supervisors, and then pass the global and server configuration settings to yaws_api:setconf/2.
- It can call yaws:start_embedded/1,2,3,4, each of which takes exactly the same arguments as the corresponding yaws_api:embedded_start_conf/1,2,3,4 function. Instead of just returning start and configuration information, however, yaws:start_embedded also starts and configures YAWS, which can be more convenient but does not allow the embedded mode application any supervision control over YAWS.

Both of these functions take care of setting the environment key embedded to true. Neither approach requires any special settings in the embedded mode application's *.app* file nor any special command-line switches to the ERLANG runtime.

For an example of how to use yaws_api:embedded_start_conf along with yaws_api:setconf, please see the files www/ybed_sup.erl and www/ybed.erl in the YAWS distribution.

Chapter 13

The config file - yaws.conf

In this section we provide a complete listing of all possible configuration file options. The configuration contains two distinct parts: a global part which affects all the virtual hosts and a server part where options for each virtual host is supplied.

13.1 Global Part

- `logdir = Directory` — All YAWS logs will be written to files in this directory. There are several different log files written by YAWS.

  - `report.log` — this is a text file that contains all error logger printouts from YAWS.
  - `<Host>.access` — for each virtual host served by YAWS, a file `<Host>.access` will be written which contains an access log in NCSA combined/XLF/ELF log format.
  - `<Host>.auth` — for each virtual host served by YAWS, a file `<Host>.auth` will be written which contains all HTTP auth related messages.
  - `trace_<YYYYMMDD_hhmmss>` - Trace files are written in this subdirectory, suffixed by the creation date.
    * `trace.<Pid>.http` - this file contains the HTTP trace if that is enabled, where `Pid` is the process id handling the TCP connection.
    * `trace.<Pid>.traffic` - this file contains the traffic trace if that is enabled, where `Pid` is the process id handling the TCP connection.

  Note that `<Host>.access` and `<Host>.auth` files will be used only if the directive `logger_mod` is not set or set to `yaws_log`.

  The default value for logdir is "."

- `ebin_dir = Directory` — This directive adds Directory to the ERLANG search path. It is possible to have several of these command in the configuration file.

- `src_dir = Directory` — This directive defines a Directory as a *source* directory. YAWS will compile all erlang modules found in this directory and all its subdirectories. The compilation occurs when the configuration is loaded or reloaded. The `include_dir` directives are used to search for includes files. Multiple `src_dir` directives may be used. There is no such directory configured by default.

- `id = String` — It is possible to run multiple Yaws servers on the same machine. We use the id of a Yaws server to control it using the different control commands such as:

  ```
  # /usr/local/bin/yaws --id foobar --stop
  ```

  To stop the Yaws server with id "foobar". Each YAWS server will write its internal data into a file called $HOME/.yaws/yaws/ID where ID is the identity of the server. Yaws also creates a file called $HOME/.yaws/yaws/ID/CTL which contains the port number where the server is listening for control commands. The default id is "default".

- `server_signature = String` — This directive sets the "Server: " output header to the custom value. The default value is "yaws/VSN, Yet Another Web Server".

- `include_dir = Directory` — This directive adds Directory to the path of directories where the ERLANG compiler searches for include files. We need to use this if we want to include *.hrl* files in our YAWS ERLANG code.

- `max_num_cached_files = Integer` — YAWS will cache small files such as commonly accessed GIF images in RAM. This directive sets a maximum number on the number of cached files. The default value is 400.

- `max_num_cached_bytes = Integer` — This directive controls the total amount of RAM which can maximally be used for cached RAM files. The default value is 1000000, 1 megabyte.

- `max_size_cached_file = Integer` — This directive sets a maximum size on the files that are RAM cached by YAWS. The default value is 8000 bytes.

- `cache_refresh_secs = Integer` — The RAM cache is used to serve pages that sit in the cache. An entry sits in cache at most cache_refresh_secs number of seconds. The default is 30. This means that when the content is updated under the docroot, that change doesn't show until 30 seconds have passed. While developing a YAWS site, it may be convenient to set this value to 0. If the debug flag (*-d*) is passed to the YAWS start script, this value is automatically set to 0.

- `trace  = traffic | http` — This enables traffic or HTTP tracing. Tracing is also possible to enable with a command line flag to YAWS.

- `auth_log  = true | false` — *Deprecated and ignored. Now, this target must be set in server part.*

- `max_connections = nolimit | Integer` — This value controls the maximum number of connections from HTTP clients into the server. This is implemented by closing the last socket if the threshold is reached.

- `keepalive_maxuses = nolimit | Integer` — Normally, YAWS does not restrict the number of times a connection is kept alive using keepalive. Setting this parameter to an integer `X` will ensure that connections are closed once they have been used `X` times. This can be a useful to guard against long-running connections collecting too much garbage in the ERLANG VM.

- `process_options = undefined | Proplist` — Set process spawn options for client acceptor processes. Options must be specified as a quoted string of either the atom `undefined` or as a proplist of valid process options. The supported options are `fullsweep_after`, `min_heap_size`, and `min_bin_vheap_size`, each taking an associated integer value. Other process options are ignored. The proplist may also be empty. See `erlang:spawn_opt/4` for details on these options.

- `large_file_chunk_size = Integer` — Set the chunk size used by YAWS to send large files. The default value is 10240.
- `large_file_sendfile = erlang | disable` — Set the version of sendfile method to use to send large files:
  - `erlang` — use *file:sendfile/5*.
  - `disable` — use *gen_tcp:send/2*.

  The default value is `erlang`.
- `acceptor_pool_size = Integer` — Set the size of the pool of cached acceptor processes. The specified value must be greater than or equal to 0. The default value is 8. Specifying a value of 0 effectively disables the process pool.
- `log_wrap_size = Integer` — The logs written by YAWS are all wrap logs, the default value at the size where they wrap around and the original gets renamed to File.old is 1000000, 1 megabyte. This value can be changed.

  If we set the value to 0 the logs will never wrap. If we want to use YAWS in combination with a more traditional log wrapper such as logrotate, set the size to 0 and YAWS will reopen the logfiles once they have be renamed/removed.
- `log_resolve_hostname = true | false` — By default the client host IP is not resolved in the access logs.
- `fail_on_bind_err = true | false` — Fail completely or not if YAWS fails to bind a listen socket Default is true.
- `soap_srv_mods = ListOfModuleSetting` — If `enable_soap` is true, a startup YAWS will invoke `yaws_soap_srv:setup()` to setup modules set here. `ModuleSetting` is either a triad like `<Mod, HandlerFun, WsdlFile>` or a tetrad like `<Mod, HandlerFun, WsdlFile, Prefix>` which specifies the prefix. A prefix will be used as argument of `yaws_soap_lib:initModel()` and then be used as a XML namespace prefix. Note, the WsdlFile here should be an absolute-path file in local file systems.

  For example, we can specify

  ```
  soap_srv_mods=<Mod1,Handler,WsdlFile1> <Mod2,Handler,Wsdl2,SpecifiedPrefix>
  ```
- `php_exe_path = Path` — *this target is deprecated and useless. use 'php_handler' target in server part instead.*

  The name of (and possibly path to) the php executable used to interpret php scripts (if allowed). Default is php-cgi.
- `copy_error_log  = true | false` — Enable or disable copying of the error log. When we run in embedded mode, there may very well be some other systems process that is responsible for writing the errorlog to a file whereas when we run in normal standalone mode, we typically want the Erlang errorlog written to a report.log file. Default value is true.
- `ysession_mod = Module` — Allows specifying a different YAWS session storage mechanism instead of an ETS table. One of the drawbacks of the default `yaws_session_server` implementation is that server side cookies are lost when the server restarts. Specifying a different module here will pass all write/read operations to this module (it must implement appropriate callbacks).

- `ysession_cookiegen = Module` — Allows specifying a different YAWS session cookie generator than the built-in default. `Module` is expected to provide a `new_cookie/0` function that returns a session cookie in the form of a list. Such a cookie generator module must be careful to return a unique cookie each time it's called.

- `ysession_idle_timeout = Integer` — Controls YAWS session idle cleanup. If a server has been idle for `ysession_idle_timeout` milliseconds, check all YAWS sessions and remove any that have timed out. The default `ysession_idle_timeout` value is 2*60*1000 (2 minutes).

- `ysession_long_timeout = Integer` — Controls YAWS session periodic cleanup. Every `ysession_long_timeout` milliseconds, check all YAWS sessions and remove any that have timed out. The default `ysession_long_timeout` value is 60*60*1000 (1 hour).

- `runmod = ModuleName` — At startup YAWS will invoke `ModuleName:start()` in a separate process. It is possible to have several runmods. This is useful if we want to reuse the Yaws startup shell script for our own application.

- `pick_first_virthost_on_nomatch = true | false` — When YAWS gets a request, it extracts the Host: header from the client request to choose a virtual server amongst all servers with the same IP/Port pair. This configuration parameter decides whether Yaws should pick the first (as defined in the yaws.conf file) if no name match or not. In real live hosting scenarios we typically want this to be false whereas in testing/development scenarios it may be convenient to set it to true. Default is true.

- `keepalive_timeout = Integer | infinity` — If the HTTP session will be kept alive (i.e., not immediately closed) it will close after the specified number of milliseconds unless a new request is received in that time. The default value is 30000. The value `infinity` is legal but not recommended.

- `subconfig = File` — Load specified config file. Absolute paths or relative ones to the configuration location are allowed. Unix-style wildcard strings can be used to include several files at once. See `filelib:wildcard/1` for details. Hidden files, starting by a dot, will be ignored. For example:

  ```
  subconfig = /etc/yaws/global.conf
  subconfig = /etc/yaws/vhosts/*.conf
  ```

  Or, relatively to the configuration location:

  ```
  subconfig = global.conf
  subconfig = vhosts/*.conf
  ```

- `subconfigdir = Directory` — Load all config files found in the specified directory. The given Directory can be an absolute path or relative to the configuration location. Hidden files, starting by a dot, will be ignored.

- `x_forwarded_for_log_proxy_whitelist = ListOfUpstreamProxyServerIps` — *This target is deprecated and will be ignored.*

- `default_type = MimeType` — Defines the default MIME type to be used where YAWS cannot determine it by its MIME types mappings. Default is *text/plain*.

- `default_charset = Charset` — Defines the default charset to be added when a response content-type is *text/**. By default, no charset is added.

- `mime_types_file = File` — Overloads the default *mime.types* file included with YAWS. This file must use the following format:

```
# Lines beginning with a '#' or a whitespace are ignored
# blank lines are also ignored
<MIME type> <space separated file extensions>
```

  The default file is located at *$PREFIX/lib/yaws/priv/mime.types*. You should not edit this file because it may be replaced when you upgrade your server.

- `add_types = ListOfTypes` — Specifies one or more mappings between MIME types and file extensions. More than one extension can be assigned to a MIME type. *ListOfTypes* is defined as follows:

```
add_types = <MimeType1, Ext> <MimeType2, Ext1 Ext2 ...> ...
```

  The mappings defined using this directive will overload all other definitions. If a file extension is defined several times, only the last one is kept. Multiple *add_types* directives may be used.

- `add_charsets = ListOfCharsets` — Specifies one or more mappings between charsets and file extensions. More than one extension can be assigned to a charset. *ListOfCharsets* is defined as follows:

```
add_charsets = <Charset1, Ext> <Charset2, Ext1 Ext2 ...> ...
```

  The mappings defined using this directive will overload all other definitions. If a file extension is defined several times, only the last one is kept. Multiple *add_charsets* directives may be used.

- `sni = disable | enable | strict` — Enables or disables the TLS SNI (Server Name Indication) support.

  When disabled (or not supported), all virtual servers in the same group (same IP/Port) must share the same SSL configuration, especially the same SSL certificate. Only the HTTP Host header will be considered to find the right virtual server.

  When enabled, SSL configuration can be different from one virtual server to another; each one can have its own SSL certificate. In this case, if a client provides a SNI hostname, it will be used to find the right virtual server. To accept the SNI information from the client, the first virtual server—the default one, see `pick_first_virthost_on_nomatch`—**must** include TLS as a permitted protocol.

  If the `sni` directive is set to *enable*, non-SNI clients are allowed. For such clients, virtual servers are selected as if Yaws did not have SNI support. If it is set to *strict*, SNI hostname is mandatory to access a SSL virtual server. But in all cases, when SNI support is enabled, if a client provides a SNI hostname, it **must** match the HTTP Host header (which is mandatory too). Note that the first virtual server (the default one) will be used for any request where the provided SNI hostname doesn't match any of virtual server names. So, it is important that the first virtual server have the most restrictive access control, otherwise clients can access restricted resources by sending a request for any unknown hostname. (This isn't actually any different from using virtual servers without SNI support.) If you're using self-signed certificates, be sure to also set the `depth` configuration variable to 0 to avoid following certificate chains.

  The `sni` directive is a global one, so if you set it to *strict*, non-SNI clients will be refused for **all** SSL groups. See `require_sni` directive from the server part to mitigate this requirement.

  Default is *disable*.

13.2 Server Part

YAWS can virthost several web servers on the same IP address as well as several web servers on different IP addresses. The only limitation here is that there can be only one server with SSL enabled per each individual IP address. Each virtual host is defined within a matching pair of `<server ServerName>` and `</server>`. The `ServerName` will be the name of the web server.

The following directives are allowed inside a server definition.

- `port = Port` — This makes the server listen on Port. Default is 8000.
- `listen = IpAddress` — This makes the server listen on `IpAddress` when virthosting several servers on the same IP/port address, if the browser doesn't send a `Host:` field, YAWS will pick the first server specified in the config file. Multiple `listen` directives may be used to specify several addresses to listen on. The default listen interface is 127.0.0.1.
- `listen_backlog = Integer` — This sets the TCP listen backlog for the server to define the maximum length the queue of pending connections may grow to. The default is 1024.
- `<listen_opts> .... </listen_opts>` — Defines extra options to be set on the listen socket and, by inheritance, on accepted sockets. See `inet:setopts/2` for details. Supported options are:
  - `Bbuffer = Integer` (default: same as `inet:setopts/2`)
  - `delay_send = true | false` (default: same as `inet:setopts/2`)
  - `linger = Integer | false` (default: same as `inet:setopts/2`)
  - `nodelay = true | false` (default: same as `inet:setopts/2`)
  - `priority = Integer` (default: same as `inet:setopts/2`)
  - `sndbuf = Integer` (default: same as `inet:setopts/2`)
  - `recbuf = Integer` (default: same as `inet:setopts/2`)
  - `send_timeout = Integer | infinity` (default: same as `inet:setopts/2`)
  - `send_timeout_close = true | false` (default: same as `inet:setopts/2`)
- `server_signature = String` — This directive sets the "Server: " output header to the custom value and overloads the global one for this virtual server.
- `subconfig = File` — Same as `subconfig` directive of the global part, but here files should only contain directives allowed in the server part.
- `subconfigdir = Directory` — Same as `subconfigdir` directive of the global part, but here files should only contain directives allowed in the server part.
- `rhost = Host[:Port]` — This forces all local redirects issued by the server to go to Host. This is useful when YAWS listens to a port which is different from the port that the user connects to. For example, running YAWS as a non-privileged user makes it impossible to listen to port 80, since that port can only be opened by a privileged user. Instead YAWS listens to a high port number port, 8000, and iptables are used to redirect traffic to port 80 to port 8000 (most NAT:ing firewalls will also do this for you).

- `rmethod = http | https` — This forces all local redirects issued by the server to use this method. This is useful when an SSL off-loader, or stunnel, is used in front of YAWS.
- `auth_log = true | false` — Enable or disable the auth log for this virtual server. Default is true.
- `access_log = true | false` — Setting this directive to false turns off traffic logging for this virtual server. The default value is true.
- `logger_mod = Module` — It is possible to set a special module that handles access and auth logging. The default is to log all web server traffic to `<Host>.access` and `<Host>.auth` files in the configured or default `logdir`.

  This module must implement the behaviour `yaws_logger`. Default value is `yaws_log`.

  The following functions should be exported:

  - `Module:open_log(ServerName, Type, LogDir)` — When YAWS is started, this function is called for this virtual server. If the initialization is successful, the function must return `{true, State}` and if an error occurred, it must return `false`.
  - `Module:close_log(ServerName, Type, State)` – This function is called for this virtual server when YAWS is stopped.
  - `Module:wrap_log(ServerName, Type, State, LogWrapSize)` — This function is used to rotate log files. It is regularly called by YAWS and must return the possibly updated internal `NewState`.
  - `Module:write_log(ServerName, Type, State, Infos)` – When it needs to log a message, YAWS will call this function. The parameter `Infos` is `{Ip, Req, InHdrs, OutHdrs, Time}` for an access log and `{Ip, Path, Item}` for an auth log, where:
    - `Ip` — IP address of the accessing client (as a tuple).
    - `Req` — The HTTP method, URI path, and HTTP version of the request (as an `#http_request{}` record).
    - `InHdrs` — The HTTP headers which were sent from the WWW client (as a `#headers{}` record).
    - `OutHdrs` — The HTTP headers sent to the WWW client (as a `#outh{}` record).
    - `Path` — The URI path of the request (as a string).
    - `Item` – The result of an authentication request. May be `{ok, User}`, `403` or `{401, Realm}`.
    - `Time` — The time taken to serve the request, in microseconds.

  For all of these callbacks, `ServerName` is the virtual server's name, `Type` is the atom `access` or `auth` and `State` is the internal state of the logger.

- `shaper = Module` — Defines a module to control access to this virtual server.

  Access can be controlled based on the IP address of the client. It is also possible to throttle HTTP requests based on the client's download rate. This module must implement the behaviour `yaws_shaper`.

  There is no such module configured by default.

- `dir_listings = true | true_nozip | false` — Setting this directive to false disallows the automatic dir listing feature of YAWS. A status code 403 Forbidden will be sent. Set to true_nozip to avoid the auto-generated all.zip entries. Default is false.

- `extra_cgi_vars = .....` — Add additional CGI or FastCGI variables. For example:

```
<extra_cgi_vars dir='/path/to/some/scripts'>
  var = val
  ...
</extra_cgi_vars>
```

- `statistics  = true | false` — Turns on/off statistics gathering for a virtual server. Default is false.

- `fcgi_app_server = Host:Port` — The hostname and TCP port number of a FastCGI application server. To specify an IPv6 address, put it inside square brackets (ex: "[::1]:9000"). The TCP port number is not optional. There is no default value.

- `fcgi_trace_protocol = true | false` — Enable or disable tracing of FastCGI protocol messages as info log messages. Disabled by default.

- `fcgi_log_app_error = true | false` — Enable or disable logging of application error messages (output to stderr and non-zero exit value). Disabled by default.

- `deflate = true | false` — Turns on or off deflate compression for a server. Default is false.

- `<deflate> ... </deflate>` — This begins and ends the deflate compression configuration for this server. The following items are allowed within a matching pair of <deflate> and </deflate> delimiters.

  - `min_compress_size = nolimit | Integer` — Defines the smallest response size that will be compressed. If nolimit is not used, the specified value must be strictly positive. The default value is nolimit.
  - `compression_level = none | default | best_compression | best_speed | 0..9` — Defines the compression level to be used. 0 (`none`), gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression. The default value is `default`.
  - `window_size = 9..15` — Specifies the zlib compression window size. It should be in the range 9 through 15. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15.
  - `mem_level = 1..9` — Specifies how much memory should be allocated for the internal compression state. `mem_level=1` uses minimum memory but is slow and reduces compression ratio; `mem_level=9` uses maximum memory for optimal speed. The default value is 8.
  - `strategy = default | filtered | huffman_only` — This parameter is used to tune the compression algorithm. See `zlib(3erl)` for more details on the `strategy` parameter. The default value is default.
  - `use_gzip_static = true | false` — If true, YAWS will try to serve precompressed versions of static files. It will look for precompressed files in the same location as original files that end in ".gz". Only files that do not fit in the cache are concerned. The default value is false.
  - `mime_types = ListOfTypes | defaults | all` — Restricts the deflate compression to particular MIME types. The special value all enable it for all types (It is a synonym of '*/*'). MIME types into `ListOfTypes` must have the form 'type/subtype' or 'type/*' (indicating all subtypes of that type). Here is an example:

```
mime_types = default image/*
mime_types = application/xml application/xhtml+xml application/rss+xml
```

By default, the following MIME types are compressed (if `deflate` is set to true):

* `text/*`
* `application/rtf`
* `application/msword`
* `application/pdf`
* `application/x-dvi`
* `application/javascript`

Multiple `mime_types` directives can be used.

- `docroot = Directory ...` — This makes the server serve all its content from `Directory`.

  It is possible to pass a space-separated list of directories as docroot. If this is the case, the various directories will be searched in order for the requested file. This also works with the ssi and yssi constructs where the full list of directories will be searched for files to ssi/yssi include. Multiple docroot directives can be used. You need at least one valid docroot, other invalid docroots are skipped with their associated auth structures.

- `auth_skip_docroot = true | false` — If true, the docroot will not be searched for `.yaws_auth` files. This is useful when the docroot is quite large and the time to search it is prohibitive when YAWS starts up. Defaults to false.

- `partial_post_size = Integer` — When a YAWS file receives large `POST`s, the amount of data received in each chunk is determined by this parameter. The default value is 10240. Setting it to nolimit is potentially dangerous.

- `dav = true | false` — Turns on the WebDAV protocol for this server. The WebDAV support adds class 1, 2 and 3 compliancy which includes:

  - XML request body parsing and multistatus responses
  - PROPFIND and PROPPATCH methods returning properties asked for
  - all RFC4918 properties, the Apache executable property plus some Microsoft extensions
  - locking mechanism (class 2 compliancy) on all destructive methods
  - If header parsing
  - Passes most litmus tests, except those concerning unknown properties

  If WebDAV is turned on, processing of `.yaws` pages is turned off. Default is false. The `dav=true` configuration option is short for:

```
runmod = yaws_runmod_lock
...
<server>
    ...
    appmods = </, yaws_appmod_dav>
<server>
```

- `tilde_expand = true|false` — If this value is set to false YAWS will never do tilde expansion. Tilde expansion takes a URL of the form `http://www.foo.com/~username` and changes it into a request where the docroot for that particular request is set to the directory `~username/public_html/`. The default value is false.

- `allowed_scripts = ListOfSuffixes` — The allowed script types for this server. Recognized are *yaws*, *cgi*, *fcgi*, *php*. Default is `allowed_scripts = yaws php cgi fcgi`.

- `tilde_allowed_scripts = ListOfSuffixes` — The allowed script types for this server when executing files in a users `public_html` folder Recognized are *yaws*, *cgi*, *fcgi*, *php*. Default is `tilde_allowed_scripts =` (i.e., empty).

- `index_files = ListOfResources` — This directive sets the list of resources to look for, when a directory is requested by the client. If the last entry begins with a '/', and none of the earlier resources are found, YAWS will perform a redirect to this uri.
  Default is `index_files = index.yaws index.html index.php`.

- `appmods = ListOfModuleNames` — If any of the names in `ListOfModuleNames` appear as components in the path for a request, the path request parsing will terminate and that module will be called. There is also an alternate syntax for specifying the appmods if we don't want our internal erlang module names to be exposed in the URL paths. We can specify

  ```
  appmods = <Path1, Module1> <Path2, Modules2> ...
  ```

  Assume for example that we have the URL `http://www.example.org/myapp/foo/bar?user=joe` while we have the module `foo` defined as an appmod, the function `foo:out(Arg)` will be invoked instead of searching the filesystems below the point `foo`.

  The `Arg` argument will have the missing path part supplied in its `appmoddata` field.

  It is also possible to exclude certain directories from appmod processing. This is particulaly interesting for '/' appmods. Here is an example:

  ```
  appmods = </, myapp exclude_paths icons js top/static>
  ```

  The above configuration will invoke the `myapp` erlang module on everything except any file found in directories `icons`, `js` and `top/static` relative to the docroot.

- `dispatchmod = DispatchModule` — Set `DispatchModule` as a server-specific request dispatching module. YAWS expects `DispatchModule` to export a `dispatch/1` function. When it receives a request, YAWS passes an `#arg{}` record to the dispatch module's `dispatch/1` function, which returns one of the following atom results:

  - `done` — this indicates the dispatch module handled the request itself and already sent the response, and YAWS should resume watching for new requests on the connection
  - `closed` — same as `done` but the `DispatchModule` also closed the connection
  - `continue` — the dispatch module has decided not to handle the request, and instead wants YAWS to perform its regular request dispatching

  Note that when `DispatchModule` handles a request itself, YAWS does not support tracing, increment statistics counters or allow traffic shaping for that request. It does however still keep track of maximum keepalive uses on the connection.

- `errormod_404 = Module` — It is possible to set a special module that handles 404 Not Found messages. The function
  `Module:out404(Arg, GC, SC)` will be invoked. The arguments are
  - `Arg` — a `#arg{}` record
  - `GC` — a `#gconf{}` record (defined in yaws.hrl)
  - `SC` — a `#sconf{}` record (defined in yaws.hrl)

  The function can and must do the same things that a normal `out/1` does.

- `errormod_401 = Module` — It is possible to set a special module that handles 401 Unauthorized messages. This can for example be used to display a login page instead. The function
  `Module:out401(Arg, Auth, Realm)` will be invoked. The arguments are
  - `Arg` — a `#arg{}` record
  - `Auth` — a `#auth{}` record
  - `Realm` — a string

  The function can and must do the same things that a normal `out/1` does.

- `errormod_crash = Module` — It is possible to set a special module that handles the HTML generation of server crash messages. The default is to display the entire formatted crash message in the browser. This is good for debugging but not in production.

  The function `Module:crashmsg(Arg, SC, Str)` will be called. The `Str` is the real crash message formatted as a string.

  The function must return, `{content,MimeType,Cont}` or `{html, Str}` or `{ehtml, Term}`. That data will be shipped to the client.

- `expires = ListOfExpires` — Controls the setting of the `Expires` HTTP header and the `max-age` directive of the `Cache-Control` HTTP header in server responses for specific MIME types. The expiration date can set to be relative to either the time the source file was last modified, or to the time of the client access. `ListOfExpires` is defined as follows:

  ```
  expires = <MimeType1, access+Seconds> <MimeType2, modify+Seconds> ...
  ```

  These HTTP headers are an instruction to the client about the document's validity and persistence. If cached, the document may be fetched from the cache rather than from the source until this time has passed. After that, the cache copy is considered "expired" and invalid, and a new copy must be obtained from the source. Here is an example:

  ```
  expires = <image/gif, access+2592000> <image/png, access+2592000>
  expires = <image/jpeg, access+2592000> <text/css, access+2592000>
  ```

- `arg_rewrite_mod = Module` — It is possible to install a module that rewrites all the `Arg #arg{}` records at an early stage in the YAWS server. This can be used to do various things such as checking a cookie, rewriting paths etc. An `arg_rewrite_mod` must export an `arg_rewrite/1` function taking and returning an `#arg{}` record. If the function wants to return a response, it must set the `#arg.state` field of its return value to an instance of the `#rewrite_response{}` record.

  The module `yaws_vdir` can be used in case you want to serve static content that is not located in your docroot. See the example at the bottom of this man page for how to use the `opaque` + `vdir` elements to instruct the `yaws_vdir` module what paths to rewrite.

- `start_mod = Module` — Defines a user-provided callback module. At startup of the server, `Module:start/1` will be called. The `#sconf{}` record (defined in `yaws.hrl`) will be used as the input argument. This makes it possible for a user application to synchronize the startup with the YAWS server as well as getting hold of user specific configuration data, see the explanation for the `<opaque>` context.

- `revproxy = Prefix Url [intercept_mod Module]` — Make YAWS a reverse proxy. `Prefix` is a path inside our own docroot and `Url` is a URL pointing to a website we want to "mount" under the `Prefix` path. For example:

  `revproxy = /tmp/foo http://www.example.org`

  This makes the `example` website appear under `/tmp/foo`.

  It is possible to have multiple reverse proxies inside the same server by supplying multiple `revproxy` directives.

  If the optional keyword `intercept_mod` and `Module` are supplied in the `revproxy` directive, then `Module` indicates a proxy request and response interception module. When the reverse proxy receives a request from a client, it will first pass `#http_request{}` and `#headers{}` records representing the client request to the intercept module's `rewrite_request/2` function. The function must return a 3-tuple of the following form:

  ```
  {ok, #http_request{}, #headers{}}
  ```

  where the record instances can be the original values passed in or new values. The reverse proxy will use these record instances when passing the request to the backend server.

  Similarly, when the backend server returns a response, the reverse proxy will call the intercept module's `rewrite_response/2` function, passing it an `#http_response{}` record and a `#headers{}` record. The function must return a 3-tuple of the following form:

  ```
  {ok, #http_response{}, #headers{}}
  ```

  where the record instances can be the original values passed in or new values. The reverse proxy will use these record instances when returning the response to the client.

  Any failure within an intercept module function results in HTTP status code 500 (Internal Server Error) being returned to the client.

  Intercept modules can use the following functions from the `yaws_api` module to retrieve, set, or delete HTTP headers from `#headers{}` records:

  - `set_header/2`, `set_header/3`
  - `get_header/2`, `get_header/3`
  - `merge_header/2`, `merge_header/3`
  - `delete_header/2`

- `fwdproxy = true|false` — Make YAWS a forward proxy. By enabling this option you can use YAWS as a proxy for outgoing web traffic, typically by configuring the proxy settings in a web-browser to explicitly target YAWS as its proxy server.

- `servername = Name` — If we're virthosting several servers and want to force a server to match specific `Host:` headers we can do this with the `servername` directive. This name doesn't necessarily have to be the same as the name inside `<server Name>` in certain NAT scenarios. Rarely used feature.

- `serveralias = ListOfNames` — This directive sets the alternate names for a virtual host. A server alias may contain wildcards:

  - '*' matches any sequence of zero or more characters
  - '?' matches one character unless that character is a period ('.')

  Multiple `serveralias` directives may be used. Here is an example:

```
<server server.domain.com>
  serveralias = server server2.domain.com server2
  serveralias = *.server.domain.com *.server?.domain.com
  ...
</server>
```

- `php_handler` — Set handler to interpret *.php* files. It can be one of the following definitions:

  - `php_handler = <cgi, Filename>` — The name of (and possibly path to) the PHP executable used to interpret PHP scripts (if allowed).
  - `php_handler = <fcgi, Host:Port>` — Use the specified FastCGI server to interpret *.php* files (if allowed).

    YAWS does not start the PHP interpreter in FastCGI mode for you. To run PHP in FastCGI mode, call it with the *-b* option. For example:

    ```
    php5-cgi -b '127.0.0.1:54321'
    ```

    This starts PHP5 in FastCGI mode listening on the local network interface. To make use of this PHP server from YAWS, specify:

    ```
    php_handler = <fcgi, 127.0.0.1:54321>
    ```

    If you need to specify an IPv6 address, use square brackets:

    ```
    php_handler = <fcgi, [::1]:54321>
    ```

    The PHP interpreter needs read access to the files it is to serve. Thus, if you run it in a different security context than YAWS itself, make sure it has access to the *.php* files.

    Please note that anyone who is able to connect to the PHP FastCGI server directly can use it to read any file to which it has read access. You should consider this when setting up a system with several mutually untrusted instances of PHP.
  - `php_handler = <extern, Module:Function | Node:Module:Function>` — Use an external handler, possibly on another node, to interpret *.php* files (if allowed).

    To interpret a *.php* file, the function `Module:Function(Arg)` will be invoked (evaluated inside an rpc call if a `Node` is specified), where `Arg` is an `#arg{}` record.

    The function must do the same things that a normal `out/1` does.

  Default value is `<cgi, "/usr/bin/php-cgi">`.

- phpfcgi = HostPortSpec — *this target is deprecated. use php_handler target in server part instead.*

  Using this directive is the same as: php_handler = <fcgi, HostPortSpec>.

- default_type = MimeType — Overloads the global default_type value for this virtual server.

- default_charset = Charset — Overloads the global default_charset value for this virtual server.

- mime_types_file = File — Overrides the global mime_type_file value for this virtual server. Mappings defined in *File* will not overload those defined by add_types directives in the global part.

- add_types = ListOfTypes — Overloads the global add_types values for this virtual server. If a mapping is defined in the global part and redefined in a server part using this directive, then it is replaced. Else it is kept.

- add_charsets = ListOfCharsets — Overloads the global add_charsets values for this virtual server. If a mapping is defined in the global part and redefined in a server part using this directive, then it is replaced. Else it is kept.

- nslookup_pref = [inet | inet6] — For fcgi servers and revproxy URLs, define the name resolution preference. For example, to perform only IPv4 name resolution, use *[inet]*. To do both IPv4 and IPv6 but try IPv6 first, use *[inet6, inet]*. Default value is *[inet]*.

- <ssl> .... </ssl> — This begins and ends an SSL configuration for this server. It's possible to virthost several SSL servers on the same IP/Port. If SNI support is disabled or not supported, they must share the same certificate configuration. In this situation, it is complicated to virthost several SSL servers on the same IP/Port since the certificate is typically bound to a domainname in the common name part of the certificate. One solution to this problem is to have a certificate with multiple subjectAltNames. If SNI support is enabled, SSL servers on the same IP/Port can have their own SSL configuration with a different SSL certificate for each one. See the global sni directive.

  The SNI support was introduced in the SSL application in Erlang/OTP 18.0. It is an extension to the TLS protocol (RFC 4366), which allows the client to include the requested hostname in the first message of its SSL handshake.

  See also http://wiki.cacert.org/VhostTaskForce#Interoperability_Test for browser compatibility.

  - keyfile = File — Specifies which file contains the private key for the certificate.
  - certfile = File — Specifies which file contains the certificate for the server.
  - cacertfile = File — A file containing trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable client CAs passed to the client when a certificate is requested.
  - dhfile = File — A file containing PEM-encoded Diffie-Hellman parameters to be used by the server if a cipher suite using Diffie-Hellman key exchange is negotiated. If not specified, default parameters are used.
  - verify = verify_none | verify_peer — Specifies the level of verification the server does on client certs. Setting verify_none means that the x509 validation will be skipped (no certificate request is sent to the client), verify_peer means that a certificate request is sent to the client (x509 validation is performed).

You might want to use `fail_if_no_peer_cert` in combination with `verify_peer`.

- `fail_if_no_peer_cert = true | false` — If verify is set to `verify_peer` and set to `true` the connection will fail if the client does not send a certificate (i.e. an empty certificate). If set to false the server will fail only if an invalid certificate is supplied (an empty certificate is considered valid).
- `depth = Int` — Specifies the depth of certificate chains the server is prepared to follow when verifying client certs. For the OTP new SSL implementation it is also used to specify how far the server (YAWS in our case) shall follow the SSL certificates we present to the clients. Hence, using self-signed certificates, we typically need to set this to 0.
- `password = String` — If the private key is encrypted on disk, this password is the 3des key to decrypt it.
- `ciphers = String` — This string specifies the SSL cipher string. The syntax of the SSL cipher string is a 4-tuple representation of the map returned by `ssl:cipher_suites/2,3`: `{#{key_exchange}, #{cipher}, #{mac}, #{prf}}`.

```
ciphers = "[{dhe_rsa,aes_256_cbc,sha,default_prf},
            {dhe_dss,aes_256_cbc,sha,default_prf}]"
```

  In older versions of Yaws, a cipher tuple lacked the `#{prf}` element. When Yaws reads a cipher of the old format from configuration, it attempts to convert it to a 4-tuple by adding `default_prf` for the `#{prf}` element. Be aware that this may not work for all ciphers; if it fails, manual intervention is needed to properly configure the ciphers in the new format.
- `eccs = String` — This string specifies the supported Elliptic Curve Cryptography (ECC). It must be a subset of *ssl:eccs()*. For PCI DSS compliance (which is the main reason why you would want to change this), set it on a single line to:

```
eccs = "[sect571r1, sect571k1, secp521r1, brainpoolP512r1, sect409k1,
         sect409r1, brainpoolP384r1, secp384r1, sect283k1, sect283r1,
         brainpoolP256r1, secp256k1, secp256r1, sect239k1, xsect233k1,
         sect233r1, secp224k1, secp224r1]"
```

- `secure_renegotiate = true | false | undefined` — Specifies whether to reject renegotiation attempt that does not live up to RFC 5746. By default `secure_renegotiate` is set to false for protocol versions that support it, i.e. secure renegotiation will be used if possible but it will fallback to unsecure renegotiation if the peer does not support RFC 5746. Set it to `undefined` to use the `ssl` module default setting to avoid errors with protocol versions that don't support it, such as TLS version 1.3. For more details, see the `ssl` manual page at http://www.erlang.org/doc/man/ssl.html
- `client_renegotiation = true | false | undefined` — Enables or disables the Erlang/OTP SSL application client renegotiation option. Defaults to true for protocol versions that support it. Set it to `undefined` to use the `ssl` module default setting to avoid errors with protocol versions that don't support it, such as TLS version 1.3. For more details, see the `ssl` manual page at http://www.erlang.org/doc/man/ssl.html
- `honor_cipher_order = true | false` — If true (the default), use the server's preference for cipher selection. If false, use the client's preference.
- `protocol_version = ProtocolList` — Specifies the list of SSL protocols that will be supported. If not set, defaults to all protocols supported by the ERLANG `ssl` application. For example, to support only TLS versions 1.3, 1.2, 1.1, and 1:

```
protocol_version = tlsv1.3, tlsv1.2, tlsv1.1, tlsv1
```

- `require_sni = true | false` — If *true*, the server will reject non-SNI clients and clients providing an unknown SNI hostname (this last remark is only relevant for the first virtual server of a SSL group). This directive is ignored if SNI support is disabled (or not supported). Default is *false*.

- `<redirect> ... </redirect>` — Defines a redirect mapping. The following items are allowed within a matching pair of `<redirect>` and `</redirect>` delimiters.

  We can have a series of redirect rules in one of the formats below:

```
Path = URL
Path = code
Path = code URL
```

  `Path` must be an url-decoded path beginning with a slash. `URL` may be either a relative URL (a path beginning with a slash), or an absolute URL. In the first case, the *scheme:hostname:port* of the current server will be added. All accesses to `Path` will be redirected to `URL/Path` (or `scheme:hostname:port/URL/Path` if `URL` is relative). `URL` must be url-encoded. Note that the original path is appended to the redirected URL.

  For example, assume we have the following redirect configuration:

```
<redirect>
  /foo = http://www.mysite.org/zapp
  /bar = /tomato.html
</redirect>
```

  Assuming this config resides on a site called `http://abc.com`, we have the following redirects:

```
http://abc.com/foo -> http://www.mysite.org/zapp/foo
http://abc.com/foo/test -> http://www.mysite.org/zapp/foo/test
http://abc.com/bar -> http://abc.com/tomato.html/bar
http://abc.com/bar/x/y/z -> http://abc.com/tomato.html/bar/x/y/z
```

  By default, YAWS will perform a 302 redirect. The HTTP status code can be changed using the `code` parameter. Note that YAWS must have knowledge of the status code value.

  - For 3xx status codes, the `URL` parameter must be present and will be used to build the new location.
  - For other status codes (1xx, 2xx, 4xx and 5xx), the `URL` can be omitted. In the absence of `URL`, YAWS will return a generic response with the specified status code.
  - Otherwise, the `URL` parameter must be a relative URL and will be used to customize the response.

  Sometimes we do not want to have the original path appended to the redirected path. To get that behaviour we specify the config with '==' instead of '='.

```
<redirect>
  /foo == http://www.mysite.org/zapp
  /bar = /tomato.html
</redirect>
```

With this configuration, a request for `http://abc.com/foo/x/y/z` simply gets redirected to `http://www.mysite.org/zapp`. This is typically used when we simply want a static redirect at some place in the docroot.

When we specify a relative URL as the target for the redirect, the redirect will be to the current http(s) server.

- `<auth> ... </auth>` — Defines an auth structure. The following items are allowed within a matching pair of <auth> and </auth> delimiters.

  - `docroot = Docroot` — If a `Docroot` is defined, this auth structure will be tested only for requests in the specified docroot. No docroot configured means all docroots. If two auth structures are defined, one with a docroot and one with no docroot, the first of both overrides the second one for requests in the configured docroot.
  - `dir = Dir` — Makes `Dir` to be controlled by WWWauthenticate headers. In order for a user to have access to WWWAuthenticate controlled directory, the user must supply a password. The `Dir` must be specified relative to the docroot. Multiple dir can be used. If no dir is set, the default value, "/", will be used.
  - `realm = Realm` — In the directory defined here, the WWW-Authenticate Realm is set to this value.
  - `authmod = AuthMod` — If an `AuthMod` is defined then `AuthMod:auth(Arg, Auth)` will be called for all access to the directory. The `auth/2` function should return one of: `true`, `false`, `{false, Realm}`, `{appmod, Mod}`. If `{appmod, Mod}` is returned then a call to `Mod:out401(Arg, Auth, Realm)` will be used to deliver the content. If `errormod_401` is defined, the call to `Mod` will be ignored. (`Mod:out(Arg)` is deprecated).

    This can, for example, be used to implement cookie authentication. The `auth()` callback would check if a valid cookie header is present, if not it would return `{appmod, ?MODULE}` and the `out401/1` function in the same module would return `{redirect_local, "/login.html"}`.
  - `user = User:Password | "User:{Algo}Hash" | "User:{Algo}$Salt$Hash"` — Inside this directory, the user `User` has access if the user supplies the password `Password` in the popup dialogue presented by the browser. It is also possible to provide a hashed password, encoded in base64. In that case, the algorithm used to hash the password must be set. `Algo` must be one of the following algorithms:

    ```
    md5 | ripemd160 | sha | sha224 | sha256 | sha384 | sha512
    ```

    It is possible to use salted hashes. If so, the `Salt` must be provided, encoded in base64. We can specify multiple users inside a single <auth> </auth> pair.
  - `pam service = pam-service` — If the item `pam` is part of the auth structure, Yaws will also try to authenticate the user using "pam" using the pam `service` indicated. Usual services are typically found under /etc/pam.d. Usual values are "system-auth" etc.

    pam authentication is performed by an Erlang port program which is typically installed as suid root by the Yaws install script.
  - `allow = all | ListOfHost` — The `allow` directive affects which hosts can access an area of the server. Access can be controlled by IP address or IP address range. If all is specified, then all hosts are allowed access, subject to the configuration of the `deny` and `order` directives. To allow only particular hosts or groups of hosts to access the server, the host can be specified in any of the following formats:

    A full IP address

```
allow = 10.1.2.3
allow = 192.168.1.104, 192.168.1.205
```

network/netmask pair

```
allow = 10.1.0.0/255.255.0.0
```

network/nnn CIDR specification

```
allow = 10.1.0.0/16
```

  - `deny = all | ListOfHost` — This directive allows access to the server to be restricted based on IP address. The arguments for the `deny` directive are identical to the arguments for the `allow` directive.
  - `order = Ordering` — The `order` directive, along with `allow` and `deny` directives, controls a three-pass access control system. The first pass processes either all `allow` or all `deny` directives, as specified by the `order` directive. The second pass parses the rest of the directives (`deny` or `allow`). The third pass applies to all requests which do not match either of the first two.

    Ordering is one of (Default value is `deny,allow`):

    * `allow,deny` — First, all `allow` directives are evaluated; at least one must match, or the request is rejected. Next, `deny` directives are evaluated. If any matches, the request is rejected. Last, any requests which do not match an `allow` or a `deny` directive are denied by default.
    * `deny,allow` — First, all `deny` directives are evaluated; if any matches, the request is denied unless it also matches an `allow` directive. Any requests which do not match any `allow` or `deny` directives are permitted.

- `<opaque> ... </opaque>` — This begins and ends an opaque configuration context for this server, where '`Key = Value`' directives can be specified. These directives are ignored by YAWS (hence the name opaque), but can be accessed as a list of tuples `{Key,Value}` stored in the `#sconf.opaque` record entry. See also the description of the `start_mod` directive.

  This mechanism can be used to pass data from a surrounding application into the individual .yaws pages.

- `strip_undefined_bindings = true|false` — Change the behavior of the `{bindings, [...]}` directive to treat all undefined keys found in returned `out/1` content as if they were defined with an empty value, resulting in all undefined bindings effectively being stripped out of returned content. By default, `strip_undefined_bindings` is false, which means undefined bindings are ignored and their text is left as is in returned content.

  This setting applies only for `out/1` content, not to static pages or other returned content.

- `<extra_response_headers> ... </extra_response_headers>` — This begins and ends a configuration context for extra response headers for this server, where directives for adding headers, erasing headers, and modules for handling extra response headers can be specified as follows:

  - `add Hdr = Value` — Add `Hdr` with value `Value` to the response, but only if the response status code is one of these values:

    * `200 OK`
    * `201 Created`

    * 204 No Content
    * 206 Partial Content
    * 301 Moved Permanently
    * 302 Found
    * 303 See Other
    * 304 Not Modified
    * 307 Temporary Redirect
    * 308 Permanent Redirect

  For any other status code, `Hdr` is not added.

- `always add Hdr = Value` — Unconditionally add `Hdr` with value `Value` to the response, regardless of the response status code.
- `erase Hdr` — Remove `Hdr` and its associated value from the response.
- `extramod = Module` — Specifies a module to call to process extra response headers. YAWS calls `Module:extra_response_headers/3` passing the following arguments:
    * Response headers — an Erlang map holding the response headers with header name strings as keys and strings as header values
    * `Arg` — an `#arg{}` record representing the request. In cases where an `extramod` module is called following the invocation of an appmod, the `#arg{}` record field `appmod_name` indicates the name of the appmod that serviced the request, allowing the `extramod` to return extra HTTP headers appropriate for that appmod.
    * `{StatusCode,Version}` — a tuple where `StatusCode` is the numeric HTTP status code for the response, and `Version` is a tuple specifying the HTTP version, e.g. `{1,1}` for HTTP 1.1.

  The `Module:extra_response_headers/3` function should return either the original header map or a modified map where headers have been added, changed, or deleted. Added headers are not subject to the status code restrictions for the `add` extra response header directive, but the function can call `yaws_api:http_extra_response_headers_add_status_codes/0` to retrieve the list of the status codes for which adding headers is normally allowed.

  For response headers that can have multiple settings, such as `Set-Cookie`, multiple values can be specified in the extra response header map by using a value of `{multi, [Value]}` where `[Value]` is a list of one or more header values. The `Set-Cookie` header is a standard special case for which YAWS converts a `multi` header into a separate `Set-Cookie` header for each value; for other headers, YAWS converts a `multi` header into a single HTTP header with a comma-separated value.

Note that extra response headers do not apply to responses returned directly by any `DispatchModule`.

• `options_asterisk_methods = Methods` — Setting `options_asterisk_methods` to a comma-separated list of HTTP `Methods` makes YAWS respond to an `OPTIONS` request that specifies a literal `*` as the target with a `200 OK` status and an `Allow` header listing the specified `Methods`. If the server configuration does not explicitly specify `options_asterisk_methods`, YAWS defaults to responding to `OPTIONS *` requests with a `200 OK` status and an `Allow` header listing these HTTP methods:

```
GET, HEAD, POST, PUT, DELETE, OPTIONS
```

RFC 7231 section 4.3 lists the standard HTTP method names:

```
GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE
```

An `options_asterisk_methods` setting can include any of these HTTP method names as well as `PATCH`. `PATCH` is not mentioned in RFC 7231 but YAWS supports it. YAWS does not implement `CONNECT`, but it supports it in `options_asterisk_methods` because it's possible to implement support for it using a `dispatchmod`.

If `options_asterisk_methods` is set to an empty value, YAWS responds to `OPTIONS *` requests with status `400 Bad Request`.

13.3 Directives included from .yaws_auth files

By default, YAWS searches each directory in the docroot for files named `.yaws_auth`. Each file is parsed by `file:consult/1`. This mechanism can be disabled by setting `auth_skip_docroot` to `true` in the YAWS configuration.

Each row in the `.yaws_auth` file must contain a term. The terms correspond to the auth structure in the YAWS config file. The allowed forms are:

```
{User, Password}.
{User, Algo, Hash}.
{User, Algo, Salt, Hash}.
{realm, String}.
{pam, Atom}.
{authmod, Atom}.
{file, String}.
{allow, all|IP_list}.
{deny, all|IP_list}.
{order, allow_deny|deny_allow}.
```

where `User`, `Password`, `Algo`, `Salt` and `Hash` are strings with double quotes.

The `.yaws_auth` file mechanism is recursive, so any subdirectories of `Dir` are also automatically protected.

The `.yaws_auth` file is never visible in a dir listing, but editor backup files such as `.yaws_auth~` *will* be visible.

13.4 Configuration Examples

The following example defines a single server on port 80.

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
</server>
```

And this example shows a similar setup but two web servers on the same IP address:

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
</server>

<server www.funky.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www_funky_org
</server>
```

When there are several virtual hosts defined for the same IP number and port, and an HTTP request arrives with a `Host` field that does not match any defined virtual host, then the one which defined "first" in the file is chosen.

An example with www-authenticate and no access logging at all.

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
        access_log = false
        <auth>
            dir = /var/yaws/www/secret
            realm = foobar
            user = jonny:verysecretpwd
            user = benny:thequestion
            user = ronny:havinganamethatendswithy
       </auth>

</server>
```

And finally a slightly more complex example with two servers on the same IP, and one SSL server on a different IP.

To force selection of the funky server if you type in for example `http://192.168.128.31/` in your browser, you can set `pick_first_virthost_on_nomatch = true` in your global configuration and rearrange the virtual server configuration order so that `www.funky.org:80` is defined first.

```
logdir = /var/log/yaws
max_num_cached_files = 8000
max_num_cached_bytes = 6000000
```

```
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
</server>

<server www.funky.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www_funky_org
</server>

<server www.funky.org>
        port = 443
        listen = 192.168.128.32
        docroot = /var/yaws/www_funky_org
        <ssl>
           keyfile = /etc/funky.key
           certfile = /etc/funky.cert
           password = gazonk
        </ssl>
</server>
```

Chapter 14

WebSocket Protocol Support

YAWS supports the WebSocket Protocol (RFC 6455), which enables two-way communication between clients and web servers. YAWS also provides support for working drafts of the WebSocket protocol, specifically drafts 10 to 17 of the hybi working group. No support for other drafts, such as those from the hixie working group, is provided.

You can find example usage of the WebSocket Protocol in the file `www/websockets_example.yaws`. This example, intended for use with any browser supporting RFC 6455, returns HTML and JavaScript that allow the client to establish a WebSocket connection to the server. These connections are handled by the code in `www/websockets_example_endpoint.yaws`, which when invoked simply establishes `examples/src/basic_echo_callback.erl` as the WebSocket callback module for the connection.

14.1 Establish a WebSocket connection

First of all, to establish a WebSocket connection, a client must send a valid HTTP Upgrade request. Then, from the server side, the YAWS script (or the appmod or whatever) should return:

```
{websocket, CallbackMod, Options}
```

where `CallbackMod` is an atom identifying the WebSocket callback module, and `Options` is a (possibly empty) list (see below for details).

From here, YAWS spawns an Erlang process to manage the WebSocket connection. Once the handshake response is returned by YAWS, the connection is established and the handling process is ready to send or receive data. If something goes wrong during this step, YAWS returns an HTTP error (400, 403 or 500 depending of the error type).

14.1.1 Supported options

The following options are available:

- `{callback, CallbackType}` — Specify the type of the callback module. `CallbackType` can be either of the following:

  - `basic` — Same as `{basic, []}`. This is the default.
  - `{basic, InitialState}` — Indicate your callback module is a basic callback module. `InitialState` is the callback's initial state for handling this client.
  - `{advanced, InitialState}` — Same as above but for an advanced callback module.

- `{origin, Origin}` — Specify the `Origin` URL from which messages will be accepted. This is useful for protecting against cross-site attack. The option defaults to `any`, meaning calls will be accepted from any origin.
- `{keepalive, KeepAliveBoolean}` — If true, YAWS will automatically send a ping message every `keepAliveTimeout` milliseconds. By default keepalive pings are disabled.
- `{keepalive_timeout, keepAliveTimeout}` — Specify the interval in milliseconds to send keepalive pings, by default 30000. Ignored if `KeepAliveBoolean` is false.
- `{keepalive_grace_period, KeepAliveGracePeriod}` — Specify the amount of time, in milliseconds, to wait after sending a keepalive ping. If no message is received within `KeepAliveGracePeriod` milliseconds, a timeout will occur. Depending on the `DropBoolean` value, a close frame is sent with the status code 1006 (if `DropBoolean` is true) or the callback module is notified (see `Module:handle_info/2` below). By default, `KeepAliveGracePeriod` is set to 2000. Ignored if `KeepAliveBoolean` is false.
- `{drop_on_timeout, DropBoolean}` — If true, a close frame is sent with the status code 1006 when a timeout occurs after a keepalive ping has been sent (see `KeepAliveGracePeriod`). Disabled by default. Ignored if `KeepAliveBoolean` is false.
- `{close_timeout, CloseTimeout}` — After sending a close frame to a client, YAWS will wait for the client acknowledgement for `CloseTimeout` milliseconds. Then it will close the underlying TCP connection. By default `CloseTimeout` is set to 5000.
- `{close_if_unmasked, CloseUnmaskedBoolean}` — If true, YAWS will reject any unmasked incoming frame by sending a close frame with the status code 1002. Disabled by default. Note: According to RFC 6455, a client must mask all frames that it sends to the server (see RFC 6455 section 5.1).
- `{max_frame_size, MaxFrameSize}` — Specify the maximum allowed size, in bytes, for received frames. By default 16MB. It is also the maximum size for unfragmented messages. This limit is checked for all types of callback module.
- `{max_message_size, MaxMsgSize}` — Specify the maximum allowed message size in bytes, by default 16MB. This limit is checked only for basic callback modules.
- `{auto_fragment_message, AutoFragBoolean}` — If true, outgoing messages will be automatically fragmented if their payload exceeds `OutFragSize` bytes. This flag is set to false by default.
- `{auto_fragment_threshold, OutFragSize}` — Specify the maximum payload size of each fragment if `AutoFragBoolean` is true. `OutFragSize` is set to 1MB by default. This setting is ignored if `AutoFragBoolean` is false.

- {hibernate_after, HibernateAfterTimeout} — If present, YAWS passes this as an option to the underlying websocket gen_server. This option causes the gen_server process to await a message for HibernateAfterTimeout milliseconds and if no message is received, the process goes into hibernation automatically. See the Erlang/OTP gen_server documentation for more details.

  Note that if the {keepalive, KeepAliveBoolean} option is also specified, the gen_server will never hibernate.

14.2 WebSocket Callback Modules

All frames received on a WebSocket connection are passed to the callback modules specified during the connection establishment by calling Module:handle_message/1 or Module:handle_message/2. depending on whether it's a basic or an advanced callback module.

14.2.1 Basic Callback Modules

When a basic callback module is used, the messages defragmentation is handled by YAWS. From the callback module point of view, all incoming messages are unfragmented. This implies that fragmented frames will be accumulated, thus basic callback modules does not support data streaming.

A basic callback module **MUST** define the stateless function Module:handle_message/1:

Module:handle_message(Message) -> Result

```
Message :: {Type, Data} | {close, Status, Reason}
Result  :: noreply | {reply, Reply} | {close, CloseReason}
  Type        :: text | binary
  Data        :: binary()
  Reply       :: {Type, Data} | #ws_frame{} |
                 [{Type, Data}] | [#ws_frame{}]
  CloseReason :: Status | {Status, Reason}
  Status      :: integer() %% RFC 6455 Status Code
  Reason      :: binary()
```

This function is called when a message is received. {text, Data} (or {binary, Data}) is the unfragmented text (or binary) message. When the client closes the connection, the callback module is notified with the message {close, Status, Reason}, where Status is the numerical status code sent by the client or the value 1000 (see RFC 6455 section 7.4.1) if the client sent no status code. For an abnormal client closure, the status code is 1006 (as specified by RFC 6455 section 7.1.5). Reason is a binary containing any text the client sent to indicate the reason for closing the socket; this binary may be empty.

If the function returns {reply, Reply}, Reply is sent to the client. It is possible to send one or more unfragmented messages by returning {Type, Data} or [{Type, Data}]. It is also possible to send one or more frames using the #ws_frame{} record instead, defined in include/yaws_api.hrl (useful to fragment messages by hand).

If the function returns noreply, nothing happens.

If the function returns `{close, CloseReason}`, the handling process closes the connection sending a close control frame to the client. `CloseReason` is used to set the status code and the (optional) close reason of the close control frame. Then the handling process terminates calling `Module:terminate(CloseReason, State)` (if defined, see below).

Because just handling messages is not enough for real applications, a basic callback module can define optional functions, mainly to manage a callback state. It can define one, some or all of the following functions:

Module:init(Args) -> Result

```
Args   :: [ReqArg, InitialState]
Result :: {ok, State} | {ok, State, Timeout} | {error, Reason}
  ReqArg       :: #arg{}
  InitialState :: term()
  State        :: term()
  Timeout      :: integer() >= 0 | infinity
  Reason       :: term()
```

If defined, this function is called to initialize the internal state of the callback module.

`ReqArg` is the `#arg{}` record supplied to the `out/1` function and `InitialState` is the term associated to the `CallbackType` described above.

If an integer timeout value is provided, it will overload the next keepalive timeout (see the `keepalive_timeout` option above). The atom `infinity` can be used to wait indefinitely. If no value is specified, the default keepalive timeout is used.

If something goes wrong during initialization, the function should return `{error, Reason}`, where `Reason` is any term.

Module:handle_open(WSState, State) -> Result

```
WSState :: #ws_state{}
State   :: term()
Result  :: {ok, NewState} | {error, Reason}
  NewState :: term()
  Reason   :: term()
```

If defined, this function is called when the connection is upgraded from HTTP to WebSocket.

`WSState` is the state of the WebSocket connection. It can be used to send messages to the client using `yaws_api:websocket_send(WSState, Message)`.

`State` is the internal state of the callback module.

If the function returns `{ok, NewState}`, the handling process will continue executing with the possibly updated internal state `NewState`.

If the function returns `{error, Reason}`, the handling process closes the connection and terminates calling `Module:terminate({error, Reason}, State)` (if defined, see below).

Module:handle_message(Message, State) -> Result

```
Message :: see Module:handle_message/1
State   :: term()
Result  :: {noreply, NewState} | {noreply, NewState, Timeout} |
           {reply, Reply} | {reply, Reply, NewState} |
           {reply, Reply, NewState, Timeout} |
           {close, CloseReason, NewState} |
           {close, CloseReason, Reply, NewState}
  NewState    :: term()
  Timeout     :: integer() >= 0 | infinity
  Reply       :: see Module:handle_message/1
  CloseReason :: see Module:handle_message/1
```

If defined, this function is called in place of `Module:handle_message/1`. The main difference with the previous version is that this one handles the internal state of the callback module.

`State` is internal state of the callback module.

See `Module:handle_message/1` for a description of the other arguments and possible return values.

Module:handle_info(Info, State) -> Result

```
Info   :: timeout | term()
State  :: term()
Result :: {noreply, NewState} | {noreply, NewState, Timeout} |
          {reply, Reply} | {reply, Reply, NewState} |
          {reply, Reply, NewState, Timeout} |
          {close, CloseReason, NewState} |
          {close, CloseReason, Reply, NewState}
  NewState    :: term()
  Timeout     :: integer() >= 0 | infinity
  Reply       :: see Module:handle_message/1
  CloseReason :: see Module:handle_message/1
```

If defined, this function is called when a timeout occurs (see `drop_on_timeout` option above) or when the handling process receives any unknown message.

`Info` is either the atom `timeout`, if a timeout has occurred, or the received message.

See `Module:handle_message/1` for a description of the other arguments and possible return values.

Module:terminate(Reason, State) -> ok

```
Reason :: Status | {Status, Text} | {error, Error}
State  :: term()
  Status :: integer() %% RFC 6455 status code
  Text   :: binary()
  Error  :: term()
```

If defined, this function is called when the handling process is about to terminate. it should be the opposite of `Module:init/1` and do any necessary cleaning up.

`Reason` is a term denoting the stop reason and `State` is the internal state of the callback module.

14.2.2 Advanced Callback Modules

Advanced callback modules should be used when automatic messages defragmentation done by YAWS is not desirable or acceptable. One could be used for example to handle data streaming over WebSockets. So, such modules should be prepared to handle frames directly (fragmented or not).

Unlike basic callback modules, Advanced ones **MUST** manage an internal state. So it **MUST** define the stateful function `Module:handle_message/2` :

Module:handle_message(Frame, State) -> Result

```
Frame   :: #ws_frame_info{} | {fail_connection, Status, Reason}
State   :: term()
Result  :: {noreply, NewState} | {noreply, NewState, Timeout} |
           {reply, Reply} | {reply, Reply, NewState} |
           {reply, Reply, NewState, Timeout} |
           {close, CloseReason, NewState} |
           {close, CloseReason, Reply, NewState}
  Status      :: integer() %% RFC 6455 status code
  Reason      :: binary()
  NewState    :: term()
  Timeout     :: integer() >= 0 | infinity
  Reply       :: see Module:handle_message/1
  CloseReason :: see Module:handle_message/1
```

This function is called when a frame is received. The `#ws_frame_info{}` record, defined in `include/yaws_api.hrl`, provides all details about this frame. `State` is the internal state of the callback module.

If an error occurs during the frame parsing, the term `{fail_connection, Status, Reason}` is passed, where `Status` is the numerical status code corresponding to the error (see RFC 6455 section 7.4.1) and `Reason` the binary containing optional information about it.

This function returns the same values as specified for the basic callback module's `Module:handle_message/2` function. See section **??** for details.

Advanced callback modules can also define the same optional functions as basic callback modules (except `Module:handle_messages/2` which is mandatory here, of course). See section **??** for details.

14.3 Record definitions

Here is the definition of records used in callback modules, defined in `include/yaws_api.hrl`:

```
%% Corresponds to the frame sections as in
%% http://tools.ietf.org/html/rfc6455#section-5.2
%% plus 'data' and 'ws_state'. Used for incoming frames.
-record(ws_frame_info, {
          fin,
          rsv,
          opcode,
          masked,
          masking_key,
          length,
          payload,
          data,        % The unmasked payload. Makes payload redundant.
          ws_state     % The ws_state after unframing this frame.
                       % This is useful for the endpoint to know what type of
                       % fragment a potentially fragmented message is.
         }).

%% Used for outgoing frames. No checks are done on the validity of a frame. This
%% is the application's responsability to send valid frames.
-record(ws_frame, {
          fin = true,
          rsv = 0,
          opcode,
          payload = <<>>
         }).

%%----------------------------------------------------------------------
%% The state of a WebSocket connection.
%% This is held by the ws owner process and passed in calls to yaws_api.
%%----------------------------------------------------------------------
-type frag_type() :: text
                   | binary
                   | none.  % The WebSocket is not expecting continuation
                            % of any fragmented message.
-record(ws_state, {
          vsn :: integer(),                    % WebSocket version number
          sock,                                % gen_tcp or gen_ssl socket
          frag_type :: frag_type()
         }).
```