

c0fd79f17d ▾

...

yaws / src / yaws_config.erl



stuart-thackray Support configuration for tlsv1.3

History

18 contributors             +6

3598 lines (3186 sloc) | 134 KB ...

```
1  %-----%
2  %%% File   : yaws_config.erl
3  %%% Author : Claes Wikstrom <klacke@bluetail.com>
4  %%% Purpose :
5  %%% Created : 16 Jan 2002 by Claes Wikstrom <klacke@bluetail.com>
6  %%%-----%
7
8  -module(yaws_config).
9  -author('klacke@bluetail.com').
10
11
12  -include("../include/yaws.hrl").
13  -include("../include/yaws_api.hrl").
14  -include("yaws_debug.hrl").
15
16  -include_lib("kernel/include/file.hrl").
17
18  -define(NEXTLINE, io_get_line(FD, '', []).).
19
20  -export([load/1,
21          make_default_gconf/2, make_default_sconf/0, make_default_sconf/3,
22          add_sconf/1,
23          add_yaws_auth/1,
24          add_yaws_soap_srv/1, add_yaws_soap_srv/2,
25          load_mime_types_module/2,
26          compile_and_load_src_dir/1,
27          search_sconf/3, search_group/3,
28          update_sconf/4, delete_sconf/3,
29          eq_sconfs/2, soft_setconf/4, hard_setconf/2,
30          can_hard_gc/2, can_soft_setconf/4,
31          can_soft_gc/2, verify_upgrade_args/2, toks/2]).
32
33  %% where to look for yaws.conf
34  paths() ->
35      case application:get_env(yaws, config) of
36          undefined ->
37              case yaws:getuid() of
38                  {ok, "0"} -> %% root
39                      [yaws_generated:etcdir() ++ "/yaws/yaws.conf"];
40                  _ -> %% developer
41                      [filename:join([yaws:home(), "yaws.conf"],
42                                     "./yaws.conf",
43                                     yaws_generated:etcdir() ++ "/yaws/yaws.conf")]
44              end;
45          {ok, File} ->
46              [File]
47      end.
48
49
50
51  %% load the config
52
53  load(E = #env{conf = false}) ->
54      case yaws:first(fun(F) -> yaws:exists(F) end, paths()) of
55          false ->
56              {error, "Can't find any config file "};
57          {ok, _, File} ->
58              load(E#env{conf = {file, File}})
59      end;
60
61  load(E) ->
62      {file, File} = E#env.conf,
63      error_logger:info_msg("Yaws: Using config file ~s~n", [File]),
64      case file:open(File, [read, {encoding, E#env.encoding}]) of
65          {ok, FD} ->
66              GC = make_default_gconf(E#env.debug, E#env.id),
67              GC1 = if E#env.traceoutput == undefined ->
68                      GC;
69                      true ->
70                          ?gc_set_tty_trace(GC, E#env.traceoutput)
71                      end,
72              GC2 = ?gc_set_debug(GC1, E#env.debug),
73              GC3 = GC2#gconf{trace = E#env.trace},
74              R = fload(FD, GC3),
75              ?Debug("LOAD(~s): ~p", [File, R]),
76              case R of
77                  {ok, GC4, Cs} ->
78                      yaws:mkdir(yaws:tmpdir()),
79                      Cs1 = add_yaws_auth(Cs),
```

```

79         add_yaws_soap_srv(GC4),
80         validate_cs(GC4, Cs1);
81     Err ->
82         Err
83     end;
84     Err ->
85         {error, ?F("Can't open config file ~s: ~p", [File, Err])}
86     end.
87
88
89 add_yaws_soap_srv(GC) when GC#gconf.enable_soap == true ->
90     add_yaws_soap_srv(GC, true);
91 add_yaws_soap_srv(_GC) ->
92     [].
93 add_yaws_soap_srv(GC, false) when GC#gconf.enable_soap == true ->
94     [{yaws_soap_srv, {yaws_soap_srv, start_link, [GC#gconf.soap_srv_mods]},
95      permanent, 5000, worker, [yaws_soap_srv]}}];
96 add_yaws_soap_srv(GC, true) when GC#gconf.enable_soap == true ->
97     Spec = add_yaws_soap_srv(GC, false),
98     case whereis(yaws_soap_srv) of
99         undefined ->
100             spawn(fun() -> supervisor:start_child(yaws_sup, hd(Spec)) end);
101         _ ->
102             ok
103     end,
104     Spec;
105 add_yaws_soap_srv(_GC, _Start) ->
106     [].
107
108
109 add_yaws_auth(#sconf{}=SC) ->
110     SC#sconf{authdirs = setup_auth(SC)};
111 add_yaws_auth(SCs) ->
112     [SC#sconf{authdirs = setup_auth(SC)} || SC <- SCs].
113
114
115 %% We search and setup www authenticate for each directory
116 %% specified as an auth directory or containing a .yaws_auth file.
117 %% These are merged with server conf.
118 setup_auth(#sconf{docroot = Docroot, xtra_docroots = XtraDocroots,
119                authdirs = Authdirs}=SC) ->
120     [begin
121         Authdirs1 = load_yaws_auth_from_docroot(D, ?sc_auth_skip_docroot(SC)),
122         Authdirs2 = load_yaws_auth_from_authdirs(Authdirs, D, []),
123         Authdirs3 = [A || A <- Authdirs1,
124                      not lists:keymember(A#auth.dir, #auth.dir, Authdirs2)],
125         Authdirs4 = ensure_auth_headers(Authdirs3 ++ Authdirs2),
126         start_pam(Authdirs4),
127         {D, Authdirs4}
128     end || D <- [Docroot|XtraDocroots] ].
129
130
131 load_yaws_auth_from_docroot(_, true) ->
132     [];
133 load_yaws_auth_from_docroot(undefined, _) ->
134     [];
135 load_yaws_auth_from_docroot(Docroot, _) ->
136     Fun = fun (Path, Acc) ->
137         %% Strip Docroot and then filename
138         SP = string:sub_string(Path, length(Docroot)+1),
139         Dir = filename:dirname(SP),
140         A = #auth{docroot=Docroot, dir=Dir},
141         case catch load_yaws_auth_file(Path, A) of
142             {ok, L} -> L ++ Acc;
143             _Other -> Acc
144         end
145     end,
146     filelib:fold_files(Docroot, "*.yaws_auth$", true, Fun, []).
147
148
149 load_yaws_auth_from_authdirs([], _, Acc) ->
150     lists:reverse(Acc);
151 load_yaws_auth_from_authdirs([Auth = #auth{dir=Dir}| Rest], Docroot, Acc) ->
152     if
153         Auth#auth.docroot /= [] andalso Auth#auth.docroot /= Docroot ->
154             load_yaws_auth_from_authdirs(Rest, Docroot, Acc);
155         Auth#auth.docroot == [] ->
156             Auth1 = Auth#auth{dir=filename:nativename(Dir)},
157             F = fun(A) ->
158                 (A#auth.docroot == Docroot andalso
159                  A#auth.dir == Auth1#auth.dir)
160             end,
161             case lists:any(F, Acc) of
162                 true ->
163                     load_yaws_auth_from_authdirs(Rest, Docroot, Acc);
164                 false ->
165                     Acc1 = Acc ++ load_yaws_auth_from_authdir(Docroot, Auth1),
166                     load_yaws_auth_from_authdirs(Rest, Docroot, Acc1)
167             end;
168         true -> %% #auth.docroot == Docroot
169             Auth1 = Auth#auth{docroot=Docroot, dir=filename:nativename(Dir)},
170             F = fun(A) ->
171                 not (A#auth.docroot == [] andalso
172                     A#auth.dir == Auth1#auth.dir)
173             end,
174             Acc1 = lists:filter(F, Acc),
175             Acc2 = Acc1 ++ load_yaws_auth_from_authdir(Docroot, Auth1),
176             load_yaws_auth_from_authdirs(Rest, Docroot, Acc2)

```

```

177     end;
178 load_yaws_auth_from_authdirs([{{Docroot, Auths}}|_], Docroot, Acc) ->
179   load_yaws_auth_from_authdirs(Auths, Docroot, Acc);
180 load_yaws_auth_from_authdirs([_| Rest], Docroot, Acc) ->
181   load_yaws_auth_from_authdirs(Rest, Docroot, Acc).
182
183
184 load_yaws_auth_from_authdir(Docroot, Auth) ->
185   Dir = case Auth#auth.dir of
186     "/" ++ R -> R;
187     _ -> Auth#auth.dir
188   end,
189   Path = filename:join([Docroot, Dir, ".yaws_auth"]),
190   case catch load_yaws_auth_file(Path, Auth) of
191     {ok, Auths} -> Auths;
192     _ -> [Auth]
193   end.
194
195
196 load_yaws_auth_file(Path, Auth) ->
197   case file:consult(Path) of
198     {ok, TermList} ->
199     error_logger:info_msg("Reading .yaws_auth ~s~n", [Path]),
200     parse_yaws_auth_file(TermList, Auth);
201     {error, _enoent} ->
202     {error, _enoent};
203     Error ->
204     error_logger:format("Bad .yaws_auth file ~s ~p~n", [Path, Error]),
205     Error
206   end.
207
208
209 ensure_auth_headers(Authdirs) ->
210   [add_auth_headers(Auth) || Auth <- Authdirs].
211
212 add_auth_headers(Auth = #auth{headers = []}) ->
213   %% Headers needs to be set
214   Realm = Auth#auth.realm,
215   Headers = yaws:make_www_authenticate_header({realm, Realm}),
216   Auth#auth{headers = Headers};
217 add_auth_headers(Auth) ->
218   Auth.
219
220
221 start_pam([]) ->
222   ok;
223 start_pam([#auth{pam = false}|T]) ->
224   start_pam(T);
225 start_pam([A|T]) ->
226   case whereis(yaws_pam) of
227     undefined -> % pam not started
228     Spec = {yaws_pam, {yaws_pam, start_link,
229       [yaws:to_list(A#auth.pam), undefined, undefined]},
230       permanent, 5000, worker, [yaws_pam]},
231     spawn(fun() -> supervisor:start_child(yaws_sup, Spec) end);
232     _ ->
233     start_pam(T)
234   end.
235
236
237 parse_yaws_auth_file([], Auth=#auth{files=[]}) ->
238   {ok, [Auth]};
239 parse_yaws_auth_file([], Auth=#auth{dir=Dir, files=Files}) ->
240   {ok, [Auth#auth{dir=filename:join(Dir, F), files=[F]} || F <- Files]};
241
242 parse_yaws_auth_file([{{realm, Realm}}|T], Auth0) ->
243   parse_yaws_auth_file(T, Auth0#auth{realm = Realm});
244
245 parse_yaws_auth_file([{{pam, Pam}}|T], Auth0)
246 when is_atom(Pam) ->
247   parse_yaws_auth_file(T, Auth0#auth{pam = Pam});
248
249 parse_yaws_auth_file([{{authmod, Authmod0}}|T], Auth0)
250 when is_atom(Authmod0) ->
251   Headers = try
252     Authmod0:get_header() ++ Auth0#auth.headers
253   catch
254     _:_ ->
255     error_logger:format("Failed to ~p:get_header() ~n",
256       [Authmod0]),
257     Auth0#auth.headers
258   end,
259   parse_yaws_auth_file(T, Auth0#auth{mod = Authmod0, headers = Headers});
260
261 parse_yaws_auth_file([{{file, File}}|T], Auth0) ->
262   Files = case File of
263     "/" ++ F -> [F|Auth0#auth.files];
264     _ -> [File|Auth0#auth.files]
265   end,
266   parse_yaws_auth_file(T, Auth0#auth{files=Files});
267
268 parse_yaws_auth_file([{{User, Password}}|T], Auth0)
269 when is_list(User), is_list(Password) ->
270   Salt = crypto:strong_rand_bytes(32),
271   Hash = crypto:hash(sha256, [Salt, Password]),
272   Users = case lists:member({User, sha256, Salt, Hash}, Auth0#auth.users) of
273     true -> Auth0#auth.users;
274     false -> [{User, sha256, Salt, Hash} | Auth0#auth.users]

```

```

275     end,
276     parse_yaws_auth_file(T, Auth0#auth(users = Users));
277
278 parse_yaws_auth_file([{{User, Algo, B64Hash}}|T], Auth0)
279 when is_list(User), is_list(Algo), is_list(B64Hash) ->
280     case parse_auth_user(User, Algo, "", B64Hash) of
281         {ok, Res} ->
282             Users = case lists:member(Res, Auth0#auth.users) of
283                 true -> Auth0#auth.users;
284                 false -> [Res | Auth0#auth.users]
285             end,
286             parse_yaws_auth_file(T, Auth0#auth(users = Users));
287         {error, Reason} ->
288             error_logger:format("Failed to parse user line ~p: ~p~n",
289                 [[User, Algo, B64Hash], Reason]),
290             parse_yaws_auth_file(T, Auth0)
291     end;
292
293 parse_yaws_auth_file([{{User, Algo, B64Salt, B64Hash}}|T], Auth0)
294 when is_list(User), is_list(Algo), is_list(B64Salt), is_list(B64Hash) ->
295     case parse_auth_user(User, Algo, B64Salt, B64Hash) of
296         {ok, Res} ->
297             Users = case lists:member(Res, Auth0#auth.users) of
298                 true -> Auth0#auth.users;
299                 false -> [Res | Auth0#auth.users]
300             end,
301             parse_yaws_auth_file(T, Auth0#auth(users = Users));
302         {error, Reason} ->
303             error_logger:format("Failed to parse user line ~p: ~p~n",
304                 [[User, Algo, B64Hash, B64Salt], Reason]),
305             parse_yaws_auth_file(T, Auth0)
306     end;
307
308 parse_yaws_auth_file([{{allow, all}}|T], Auth0) ->
309     Auth1 = case Auth0#auth.acl of
310         none -> Auth0#auth{acl={all, [], deny_allow}};
311         {_,D,0} -> Auth0#auth{acl={all, D, 0}}
312     end,
313     parse_yaws_auth_file(T, Auth1);
314
315 parse_yaws_auth_file([{{allow, IPs}}|T], Auth0) when is_list(IPs) ->
316     Auth1 = case Auth0#auth.acl of
317         none ->
318             AllowIPs = parse_auth_ips(IPs, []),
319             Auth0#auth{acl={AllowIPs, [], deny_allow}};
320         {all, _, _} ->
321             Auth0;
322         {AllowIPs, DenyIPs, Order} ->
323             AllowIPs2 = parse_auth_ips(IPs, []) ++ AllowIPs,
324             Auth0#auth{acl={AllowIPs2, DenyIPs, Order}}
325     end,
326     parse_yaws_auth_file(T, Auth1);
327
328 parse_yaws_auth_file([{{deny, all}}|T], Auth0) ->
329     Auth1 = case Auth0#auth.acl of
330         none -> Auth0#auth{acl={[], all, deny_allow}};
331         {A,_,0} -> Auth0#auth{acl={A, all, 0}}
332     end,
333     parse_yaws_auth_file(T, Auth1);
334
335 parse_yaws_auth_file([{{deny, IPs}}|T], Auth0) when is_list(IPs) ->
336     Auth1 = case Auth0#auth.acl of
337         none ->
338             DenyIPs = parse_auth_ips(IPs, []),
339             Auth0#auth{acl={[], DenyIPs, deny_allow}};
340         {_, all, _} ->
341             Auth0;
342         {AllowIPs, DenyIPs, Order} ->
343             DenyIPs2 = parse_auth_ips(IPs, []) ++ DenyIPs,
344             Auth0#auth{acl={AllowIPs, DenyIPs2, Order}}
345     end,
346     parse_yaws_auth_file(T, Auth1);
347
348 parse_yaws_auth_file([{{order, 0}}|T], Auth0)
349 when 0 == allow_deny; 0 == deny_allow ->
350     Auth1 = case Auth0#auth.acl of
351         none -> Auth0#auth{acl={[], [], 0}};
352         {A,D,_} -> Auth0#auth{acl={A, D, 0}}
353     end,
354     parse_yaws_auth_file(T, Auth1).
355
356
357
358 %% Create mime_types.erl, compile it and load it. If everything is ok,
359 %% reload groups.
360 %%
361 %% If an error occurred, the previously-loaded version (the first time, it's the
362 %% static version) is kept.
363 load_mime_types_module(GC, Groups) ->
364     GInfo = GC#gconf.mime_types_info,
365     SInfos = [{(SC#sconf.servername, SC#sconf.port), SC#sconf.mime_types_info}
366         || SC <- lists:flatten(Groups),
367         SC#sconf.mime_types_info /= undefined],
368
369     case {is_dir(yaws:id_dir(GC#gconf.id)), is_dir(yaws:tmpdir("tmp"))} of
370         {true, _} ->
371             File = filename:join(yaws:id_dir(GC#gconf.id), "mime_types.erl"),
372             load_mime_types_module(File, GInfo, SInfos);

```

```

373     (_, true) ->
374         File = filename:join(yaws:tmpdir("/tmp"), "mime_types.erl"),
375         load_mime_types_module(File, GInfo, SInfos);
376     ->
377         error_logger:format("Cannot write module mime_types.erl-n"
378             "Keep the previously-loaded version-n", [])
379 end,
380 lists:map(fun(Gp) ->
381     [begin
382         F = fun(X) when is_atom(X) -> X;
383           (X) -> element(1, mime_types:t(SC, X))
384         end,
385         TAS = SC#sconf.tilde_allowed_scripts,
386         AS = SC#sconf.allowed_scripts,
387         SC#sconf{tilde_allowed_scripts=lists:map(F, TAS),
388             allowed_scripts=lists:map(F, AS)}
389         end || SC <- Gp]
390     end, Groups).
391
392 load_mime_types_module(_, undefined, []) ->
393     ok;
394 load_mime_types_module(File, undefined, SInfos) ->
395     load_mime_types_module(File, #mime_types_info{}, SInfos);
396 load_mime_types_module(File, GInfo, SInfos) ->
397     case mime_type_c:generate(File, GInfo, SInfos) of
398     ok ->
399         case compile:file(File, [binary]) of
400         {ok, ModName, Binary} ->
401             case code:load_binary(ModName, [], Binary) of
402             {module, ModName} ->
403                 ok;
404             {error, What} ->
405                 error_logger:format(
406                     "Cannot load module '~p': ~p-n"
407                     "Keep the previously-loaded version-n",
408                     [ModName, What]
409                 )
410             end;
411         ->
412             error_logger:format("Compilation of '~p' failed-n"
413                 "Keep the previously-loaded version-n",
414                 [File])
415         end;
416     {error, Reason} ->
417         error_logger:format("Cannot write module ~p: ~p-n"
418             "Keep the previously-loaded version-n",
419             [File, Reason])
420     end.
421
422 %% Compile modules found in the configured source directories, recursively.
423 compile_and_load_src_dir(GC) ->
424     Incs = lists:map(fun(Dir) -> {i, Dir} end, GC#gconf.include_dir),
425     Opts = [binary, return] ++ Incs,
426     lists:foreach(fun(D) -> compile_and_load_src_dir([], [D], Opts) end,
427         GC#gconf.src_dir).
428
429 compile_and_load_src_dir(_Dir, [], _Opts) ->
430     ok;
431 compile_and_load_src_dir(Dir, [Entry0|Rest], Opts) ->
432     Entry1 = case Dir of
433     [] -> Entry0;
434     _ -> filename:join(Dir, Entry0)
435     end,
436     case filelib:is_dir(Entry1) of
437     true ->
438         case file:list_dir(Entry1) of
439         {ok, Files} ->
440             compile_and_load_src_dir(Entry1, Files, Opts);
441         {error, Reason} ->
442             error_logger:format("Failed to compile modules in ~p: ~s-n",
443                 [Entry1, file:format_error(Reason)])
444         end;
445     false ->
446         case filename:extension(Entry0) of
447         ".erl" -> compile_module_src_dir(Entry1, Opts);
448         _ -> ok
449         end
450     end,
451     compile_and_load_src_dir(Dir, Rest, Opts).
452
453 compile_module_src_dir(File, Opts) ->
454     case catch compile:file(File, Opts) of
455     {ok, Mod, Bin} ->
456         error_logger:info_msg("Compiled ~p-n", [File]),
457         load_src_dir(File, Mod, Bin);
458     {ok, Mod, Bin, []} ->
459         error_logger:info_msg("Compiled ~p [0 Errors - 0 Warnings]-n", [File]),
460         load_src_dir(File, Mod, Bin);
461     {ok, Mod, Bin, Warnings} ->
462         WsMsg = [format_compile_warns(W, []) || W <- Warnings],
463         error_logger:warning_msg("Compiled ~p [~p Errors - ~p Warnings]-n-s",
464             [File, 0, length(WsMsg), WsMsg]),
465         load_src_dir(File, Mod, Bin);
466     {error, [], Warnings} ->
467         WsMsg = [format_compile_warns(W, []) || W <- Warnings],
468         error_logger:format("Failed to compile ~p "

```

```

471         "[~p Errors - ~p Warnings]~n~s"
472         "**** warnings being treated as errors~n",
473         [File,0,length(WsMsg),WsMsg]);
474     {error, Errors, Warnings} ->
475         WsMsg = [format_compile_warns(W,[]) || W <- Warnings],
476         EsMsg = [format_compile_errs(E,[]) || E <- Errors],
477         error_logger:format("Failed to compile ~p "
478             "[~p Errors - ~p Warnings]~n~s~s",
479             [File,length(EsMsg),length(WsMsg),EsMsg,WsMsg]);
480     error ->
481         error_logger:format("Failed to compile ~p~n", [File]);
482     {'EXIT', Reason} ->
483         error_logger:format("Failed to compile ~p: ~p~n", [File, Reason])
484 end.
485
486
487 load_src_dir(File, Mod, Bin) ->
488     case code:load_binary(Mod, File, Bin) of
489     {module, Mod} -> ok;
490     {error, Reason} -> error_logger:format("Cannot load module ~p: ~p~n",
491         [Mod, Reason])
492 end.
493
494 format_compile_warns(%, [], Acc) ->
495     lists:reverse(Acc);
496 format_compile_warns({File, [{L,M,E}|Rest]}, Acc) ->
497     Msg = io_lib:format(" ~s:~w: Warning: ~s~n", [File,L,M:format_error(E)]),
498     format_compile_warns({File, Rest}, [Msg|Acc]).
499
500 format_compile_errs(%, [], Acc) ->
501     lists:reverse(Acc);
502 format_compile_errs({File, [{L,M,E}|Rest]}, Acc) ->
503     Msg = io_lib:format(" ~s:~w: ~s~n", [File,L,M:format_error(E)]),
504     format_compile_errs({File, Rest}, [Msg|Acc]).
505
506
507
508 %% This is the function that arranges sconfs into
509 %% different server groups
510 validate_cs(GC, Cs) ->
511     L = lists:map(fun(#sconf{listen=IP0}=SC0) ->
512         SC = case is_tuple(IP0) of
513             false ->
514                 {ok, IP} = inet_parse:address(IP0),
515                 SC0#sconf{listen=IP};
516             true ->
517                 SC0
518         end,
519         {{SC#sconf.listen, SC#sconf.port}, SC}
520     end, Cs),
521     L2 = lists:map(fun(X) -> element(2, X) end, lists:keysort(1,L)),
522     L3 = arrange(L2, start, [], []),
523     case validate_groups(GC, L3) of
524     ok ->
525         {ok, GC, L3};
526     Err ->
527         Err
528     end.
529
530
531 validate_groups(_, []) ->
532     ok;
533 validate_groups(GC, [H|T]) ->
534     case (catch validate_group(GC, H)) of
535     ok ->
536         validate_groups(GC, T);
537     Err ->
538         Err
539     end.
540
541 validate_group(GC, List) ->
542     [SC0|SCs] = List,
543
544     %% all servers with the same IP/Port must share the same tcp configuration
545     case lists:all(fun(SC) ->
546         lists:keyfind(listen_opts, 1, SC#sconf.soptions) ==
547         lists:keyfind(listen_opts, 1, SC0#sconf.soptions)
548     end, SCs) of
549     true ->
550         ok;
551     false ->
552         throw({error, ?F("Servers in the same group must share the same tcp"
553             " configuration: ~p", [SC0#sconf.servername])})
554     end,
555
556     %% If the default servers (the first one) is not an SSL server:
557     %% all servers with the same IP/Port must be non-SSL server
558     %% If SNI is disabled or not supported:
559     %% all servers with the same IP/Port must share the same SSL config
560     %% If SNI is enabled:
561     %% TLS protocol must be supported by the default servers (the first one)
562     if
563         SC0#sconf.ssl == undefined ->
564             case lists:all(fun(SC) -> SC#sconf.ssl == SC0#sconf.ssl end, SCs) of
565             true -> ok;
566             false ->
567                 throw({error, ?F("All servers in the same group than"
568                     " ~p must have no SSL configuration",

```

```

569         [SC0#sconf.servername]))}
570     end;
571     GC#gconf.sni == disable ->
572     case lists:all(fun(SC) -> SC#sconf.ssl == SC0#sconf.ssl end, SCs) of
573     true -> ok;
574     false ->
575         throw({error, ?F("SNI is disabled, all servers in the same"
576             " group than ~p must share the same ssl"
577             " configuration",
578             [SC0#sconf.servername]))}
579     end;
580
581     true ->
582     Vs = case (SC0#sconf.ssl)#ssl.protocol_version of
583         undefined -> proplists:get_value(available,ssl:versions());
584         L -> L
585     end,
586     F = fun(V) -> lists:member(V, ['tlsv1.3','tlsv1.2','tlsv1.1',tlsv1]) end,
587     case lists:any(F, Vs) of
588     true -> ok;
589     false ->
590         throw({error, ?F("SNI is enabled, the server ~p must enable"
591             " TLS protocol", [SC0#sconf.servername]))}
592     end
593 end,
594
595 %% all servernames in a group must be unique
596 SN = lists:sort([yaws:to_lower(X#sconf.servername) || X <- List]),
597 no_two_same(SN).
598
599 no_two_same([H,H|_]) ->
600     throw({error,
601         ?F("Two servers in the same group cannot have same name ~p",[H]))};
602 no_two_same([_H|T]) ->
603     no_two_same(T);
604 no_two_same([]) ->
605     ok.
606
607
608
609 arrange([C|Tail], start, [], B) ->
610     C1 = set_server(C),
611     arrange(Tail, {in, C1}, [C1], B);
612 arrange([], _, [], B) ->
613     B;
614 arrange([], _, A, B) ->
615     [lists:reverse(A) | B];
616 arrange([C|Tail], {in, C0}, A, B) ->
617     C1 = set_server(C),
618     if
619     C1#sconf.listen == C0#sconf.listen,
620     C1#sconf.port == C0#sconf.port ->
621         arrange(Tail, {in, C0}, [C1|A], B);
622     true ->
623         arrange(Tail, {in, C1}, [C1], [lists:reverse(A)|B])
624     end.
625
626
627 set_server(SC) ->
628     SC1 = if
629         SC#sconf.port == 0 ->
630             {ok, P} = yaws:find_private_port(),
631             SC#sconf{port=P};
632         true ->
633             SC
634     end,
635     case {SC1#sconf.ssl, SC1#sconf.port, ?sc_has_add_port(SC1)} of
636     {undefined, 80, _} ->
637         SC1;
638     {undefined, Port, true} ->
639         add_port(SC1, Port);
640     {_SSL, 443, _} ->
641         SC1;
642     {_SSL, Port, true} ->
643         add_port(SC1, Port);
644     {__,_} ->
645         SC1
646     end.
647
648
649 add_port(SC, Port) ->
650     case string:tokens(SC#sconf.servername, ":") of
651     [Srv, Prt] ->
652         case (catch list_to_integer(Prt)) of
653         {'EXIT', _} ->
654             SC#sconf{servername =
655                 Srv ++ [$.|integer_to_list(Port)]};
656         _Int ->
657             SC
658         end;
659     [Srv] ->
660         SC#sconf{servername = Srv ++ [$.|integer_to_list(Port)]}
661     end.
662
663
664 make_default_gconf(Debug, Id) ->
665     Y = yaws_dir(),
666     Flags = (?GC_COPY_ERRLOG bor ?GC_FAIL_ON_BIND_ERR bor

```

```

667         ?GC_PICK_FIRST_VIRTHOST_ON_NOMATCH),
668     #gconf{yaws_dir = Y,
669         ebin_dir = [filename:join([Y, "examples/ebin"])],
670         include_dir = [filename:join([Y, "examples/include"])],
671         trace = false,
672         logdir = ".",
673         cache_refresh_secs = if
674             Debug == true ->
675                 0;
676             true ->
677                 30
678         end,
679         flags = if Debug -> Flags bor ?GC_DEBUG;
680             true -> Flags
681         end,
682
683         yaws = "Yaws " ++ yaws_generated:version(),
684         id = Id
685     }.
686
687 %% Keep this function for backward compatibility. But no one is supposed to use
688 %% it (yaws_config is an internal module, its api is private).
689 make_default_sconf() ->
690     make_default_sconf([], undefined, undefined).
691
692 make_default_sconf([], Servername, Port) ->
693     make_default_sconf(filename:join([yaws_dir(), "www"]), Servername, Port);
694 make_default_sconf(DocRoot, undefined, Port) ->
695     make_default_sconf(DocRoot, "localhost", Port);
696 make_default_sconf(DocRoot, Servername, undefined) ->
697     make_default_sconf(DocRoot, Servername, 8000);
698 make_default_sconf(DocRoot, Servername, Port) ->
699     AbsDocRoot = filename:absname(DocRoot),
700     case is_dir(AbsDocRoot) of
701         true ->
702             set_server(#sconf{port=Port, servername=Servername,
703                 listen={127,0,0,1}, docroot=AbsDocRoot});
704         false ->
705             throw({error, ?F("Invalid docroot: directory ~s does not exist",
706                 [AbsDocRoot])});
707     end.
708
709
710 yaws_dir() ->
711     yaws:get_app_dir().
712
713 string_to_host_and_port(String) ->
714     HostPortRE = "^(?:\\[[^\\]]+\\]|\\[[:]+\\]):([0-9]+)$",
715     REOptions = [{capture, all_but_first, list}],
716     case re:run(String, HostPortRE, REOptions) of
717         {match, [IPv6, HostOrIPv4, Port]} ->
718             case string_to_integer(Port) of
719                 {Integer, []} when Integer >= 0, Integer <= 65535 ->
720                     case IPv6 of
721                         "" -> {ok, HostOrIPv4, Integer};
722                         _ -> {ok, IPv6, Integer}
723                     end;
724                 _Else ->
725                     {error, ?F("~p is not a valid port number", [Port])}
726             end;
727         nomatch ->
728             {error, ?F("bad host and port specifier, expected HOST:PORT; "
729                 "use [IP]:PORT for IPv6 address", [])}
730     end.
731
732 string_to_node_mod_fun(String) ->
733     case string:tokens(String, ".:") of
734         [Node, Mod, Fun] ->
735             {ok, list_to_atom(Node), list_to_atom(Mod), list_to_atom(Fun)};
736         [Mod, Fun] ->
737             {ok, list_to_atom(Mod), list_to_atom(Fun)};
738         _ ->
739             {error, ?F("bad external module specifier, "
740                 "expected NODE:MODULE:FUNCTION or MODULE:FUNCTION", [])}
741     end.
742
743
744
745 %% two states, global, server
746 fload(FD, GC) ->
747     case catch fload(FD, GC, [], 1, ?NEXTLINE) of
748         {ok, GC1, Cs} -> {ok, GC1, lists:reverse(Cs)};
749         Err -> Err
750     end.
751
752
753 fload(FD, GC, Cs, _Lno, eof) ->
754     file:close(FD),
755     {ok, GC, Cs};
756
757 fload(FD, GC, Cs, Lno, Chars) ->
758     case toks(Lno, Chars) of
759         [] ->
760             fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
761
762         ["subconfig", '=', Name] ->
763             case subconfig_files(FD, Name, Lno) of
764                 {ok, Files} ->

```



```

765         case fload_subconfigfiles(Files, global, GC, Cs) of
766         {ok, GC1, Cs1} ->
767             fload(FD, GC1, Cs1, Lno+1, ?NEXTLINE);
768         Err ->
769             Err
770         end;
771     Err ->
772     Err
773 end;
774
775 ["subconfigdir", '=', Name] ->
776     case subconfigdir(FD, Name, Lno) of
777     {ok, Files} ->
778         case fload_subconfigfiles(Files, global, GC, Cs) of
779         {ok, GC1, Cs1} ->
780             fload(FD, GC1, Cs1, Lno+1, ?NEXTLINE);
781         Err ->
782             Err
783         end;
784     Err ->
785     Err
786 end;
787
788 ["trace", '=', Bstr] when GC#gconf.trace == false ->
789     case Bstr of
790     "traffic" ->
791         fload(FD, GC#gconf{trace = {true, traffic}}, Cs,
792             Lno+1, ?NEXTLINE);
793     "http" ->
794         fload(FD, GC#gconf{trace = {true, http}}, Cs,
795             Lno+1, ?NEXTLINE);
796     "false" ->
797         fload(FD, GC#gconf{trace = false}, Cs, Lno+1, ?NEXTLINE);
798     _ ->
799         {error, ?F("Expect false|http|traffic at line ~w",[Lno])}
800     end;
801
802 ["trace", '=', _Bstr] ->
803     %% don't overwrite setting from commandline
804     fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
805
806 ["logdir", '=', Logdir] ->
807     Dir = case Logdir of
808     "+" ++ D ->
809         D1 = filename:absname(D),
810         %% try to make the log directory if it doesn't exist
811         yaws:mkdir(D1),
812         D1;
813     _ ->
814         filename:absname(Logdir)
815     end,
816     case is_dir(Dir) of
817     true ->
818         put(logdir, Dir),
819         fload(FD, GC#gconf{logdir = Dir}, Cs, Lno+1, ?NEXTLINE);
820     false ->
821         {error, ?F("Expect directory at line ~w (logdir ~s)",
822             [Lno, Dir])}
823     end;
824
825 ["ebin_dir", '=', Ebindir] ->
826     Dir = filename:absname(Ebindir),
827     case warn_dir("ebin_dir", Dir) of
828     true ->
829         fload(FD, GC#gconf{ebin_dir = [Dir|GC#gconf.ebin_dir]}, Cs,
830             Lno+1, ?NEXTLINE);
831     false ->
832         fload(FD, GC, Cs, Lno+1, ?NEXTLINE)
833     end;
834
835 ["src_dir", '=', Srcdir] ->
836     Dir = filename:absname(Srcdir),
837     case warn_dir("src_dir", Dir) of
838     true ->
839         fload(FD, GC#gconf{src_dir = [Dir|GC#gconf.src_dir]}, Cs,
840             Lno+1, ?NEXTLINE);
841     false ->
842         fload(FD, GC, Cs, Lno+1, ?NEXTLINE)
843     end;
844
845 ["runmod", '=', Mod0] ->
846     Mod = list_to_atom(Mod0),
847     fload(FD, GC#gconf{runmods = [Mod|GC#gconf.runmods]}, Cs,
848         Lno+1, ?NEXTLINE);
849
850 ["enable_soap", '=', Bool] ->
851     if (Bool == "true") ->
852         fload(FD, GC#gconf{enable_soap = true}, Cs,
853             Lno+1, ?NEXTLINE);
854     true ->
855         fload(FD, GC#gconf{enable_soap = false}, Cs,
856             Lno+1, ?NEXTLINE)
857     end;
858
859 ["soap_srv_mods", '=' | SoapSrvMods] ->
860     case parse_soap_srv_mods(SoapSrvMods, []) of
861     {ok, L} ->
862         fload(FD, GC#gconf{soap_srv_mods = L}, Cs,

```

```

863         Lno+1, ?NEXTLINE);
864     {error, Str} ->
865         {error, ?F("~s at line ~w", [Str, Lno])}
866     end;
867
868 ["max_connections", '=', Int] ->
869     case (catch list_to_integer(Int)) of
870     I when is_integer(I) ->
871         fload(FD, GC#gconf{max_connections = I}, Cs,
872             Lno+1, ?NEXTLINE);
873     _ when Int == "noLimit" ->
874         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
875     _ ->
876         {error, ?F("Expect integer at line ~w", [Lno])}
877     end;
878
879 ["process_options", '=', POpts] ->
880     case parse_process_options(POpts) of
881     {ok, ProcList} ->
882         fload(FD, GC#gconf{process_options=ProcList}, Cs,
883             Lno+1, ?NEXTLINE);
884     {error, Str} ->
885         {error, ?F("~s at line ~w", [Str, Lno])}
886     end;
887
888 ["large_file_chunk_size", '=', Int] ->
889     case (catch list_to_integer(Int)) of
890     I when is_integer(I) ->
891         fload(FD, GC#gconf{large_file_chunk_size = I}, Cs,
892             Lno+1, ?NEXTLINE);
893     _ ->
894         {error, ?F("Expect integer at line ~w", [Lno])}
895     end;
896
897 ["large_file_sendfile", '=', Method] ->
898     case set_sendfile_flags(GC, Method) of
899     {ok, GC1} ->
900         fload(FD, GC1, Cs, Lno+1, ?NEXTLINE);
901     {error, Str} ->
902         {error, ?F("~s at line ~w", [Str, Lno])}
903     end;
904
905 ["acceptor_pool_size", '=', Int] ->
906     case catch list_to_integer(Int) of
907     I when is_integer(I), I >= 0 ->
908         fload(FD, GC#gconf{acceptor_pool_size = I}, Cs,
909             Lno+1, ?NEXTLINE);
910     _ ->
911         {error, ?F("Expect integer >= 0 at line ~w", [Lno])}
912     end;
913
914 ["log_wrap_size", '=', Int] ->
915     case (catch list_to_integer(Int)) of
916     I when is_integer(I) ->
917         fload(FD, GC#gconf{log_wrap_size = I}, Cs,
918             Lno+1, ?NEXTLINE);
919     _ ->
920         {error, ?F("Expect integer at line ~w", [Lno])}
921     end;
922
923 ["log_resolve_hostname", '=', Bool] ->
924     case is_bool(Bool) of
925     {true, Val} ->
926         fload(FD, ?gc_log_set_resolve_hostname(GC, Val), Cs,
927             Lno+1, ?NEXTLINE);
928     false ->
929         {error, ?F("Expect true|false at line ~w", [Lno])}
930     end;
931
932 ["fail_on_bind_err", '=', Bool] ->
933     case is_bool(Bool) of
934     {true, Val} ->
935         fload(FD, ?gc_set_fail_on_bind_err(GC, Val), Cs,
936             Lno+1, ?NEXTLINE);
937     false ->
938         {error, ?F("Expect true|false at line ~w", [Lno])}
939     end;
940
941
942 ["include_dir", '=', Incdir] ->
943     Dir = filename:absname(Incdir),
944     case warn_dir("include_dir", Dir) of
945     true ->
946         fload(FD, GC#gconf{include_dir= [Dir|GC#gconf.include_dir]},
947             Cs, Lno+1, ?NEXTLINE);
948     false ->
949         fload(FD, GC, Cs, Lno+1, ?NEXTLINE)
950     end;
951
952
953 ["mnesia_dir", '=', Mnesiadir] ->
954     Dir = filename:absname(Mnesiadir),
955     case is_dir(Dir) of
956     true ->
957         put(mnesiadir, Dir),
958         fload(FD, GC#gconf{mnesia_dir = Dir}, Cs, Lno+1, ?NEXTLINE);
959     false ->
960         {error, ?F("Expect mnesia directory at line ~w", [Lno])}

```

```

961     end;
962
963 ["tmpdir", '=', _TmpDir] ->
964     %% ignore
965     error_logger:format(
966         "tmpdir in yaws.conf is no longer supported - ignoring\n",[]
967     ),
968     fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
969
970 ["keepalive_timeout", '=', Val] ->
971     %% keep this bugger for backward compat for a while
972     case (catch list_to_integer(Val)) of
973     I when is_integer(I) ->
974         fload(FD, GC#gconf{keepalive_timeout = I}, Cs,
975             Lno+1, ?NEXTLINE);
976     _ when Val == "infinity" ->
977         fload(FD, GC#gconf{keepalive_timeout = infinity}, Cs,
978             Lno+1, ?NEXTLINE);
979     _ ->
980         {error, ?F("Expect integer at line ~w", [Lno])}
981     end;
982
983 ["keepalive_maxuses", '=', Int] ->
984     case (catch list_to_integer(Int)) of
985     I when is_integer(I) ->
986         fload(FD, GC#gconf{keepalive_maxuses = I}, Cs,
987             Lno+1, ?NEXTLINE);
988     _ when Int == "nolimit" ->
989         %% nolimit is the default
990         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
991     _ ->
992         {error, ?F("Expect integer at line ~w", [Lno])}
993     end;
994
995 ["php_exe_path", '=', PhpPath] ->
996     error_logger:format(
997         "'php_exe_path' is deprecated, use 'php_handler' instead\n",
998         []),
999     case is_file(PhpPath) of
1000     true ->
1001         fload(FD, GC#gconf{phpexe = PhpPath}, Cs, Lno+1, ?NEXTLINE);
1002     false ->
1003         {error, ?F("Expect executable file at line ~w", [Lno])}
1004     end;
1005
1006 ["read_timeout", '=', _Val] ->
1007     %% deprecated, don't use
1008     error_logger:format(
1009         "read_timeout in yaws.conf is no longer supported - ignoring\n",[]
1010     ),
1011     fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1012
1013 ["max_num_cached_files", '=', Val] ->
1014     case (catch list_to_integer(Val)) of
1015     I when is_integer(I) ->
1016         fload(FD, GC#gconf{max_num_cached_files = I}, Cs,
1017             Lno+1, ?NEXTLINE);
1018     _ ->
1019         {error, ?F("Expect integer at line ~w", [Lno])}
1020     end;
1021
1022
1023 ["max_num_cached_bytes", '=', Val] ->
1024     case (catch list_to_integer(Val)) of
1025     I when is_integer(I) ->
1026         fload(FD, GC#gconf{max_num_cached_bytes = I}, Cs,
1027             Lno+1, ?NEXTLINE);
1028     _ ->
1029         {error, ?F("Expect integer at line ~w", [Lno])}
1030     end;
1031
1032
1033 ["max_size_cached_file", '=', Val] ->
1034     case (catch list_to_integer(Val)) of
1035     I when is_integer(I) ->
1036         fload(FD, GC#gconf{max_size_cached_file = I}, Cs,
1037             Lno+1, ?NEXTLINE);
1038     _ ->
1039         {error, ?F("Expect integer at line ~w", [Lno])}
1040     end;
1041
1042 ["cache_refresh_secs", '=', Val] ->
1043     case (catch list_to_integer(Val)) of
1044     I when is_integer(I), I >= 0 ->
1045         fload(FD, GC#gconf{cache_refresh_secs = I}, Cs,
1046             Lno+1, ?NEXTLINE);
1047     _ ->
1048         {error, ?F("Expect 0 or positive integer at line ~w",[Lno])}
1049     end;
1050
1051
1052 ["copy_error_log", '=', Bool] ->
1053     case is_bool(Bool) of
1054     {true, Val} ->
1055         fload(FD, ?gc_set_copy_errlog(GC, Val), Cs,
1056             Lno+1, ?NEXTLINE);
1057     false ->
1058         {error, ?F("Expect true|false at line ~w", [Lno])}

```

```

1059         end;
1060
1061
1062     ["auth_log", '=', Bool] ->
1063         error_logger:format(
1064             "auth_log' global variable is deprecated and ignored."
1065             " it is now a per-server variable", []),
1066         case is_bool(Bool) of
1067             {true, _Val} ->
1068                 fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1069             false ->
1070                 {error, ?F("Expect true|false at line ~w", [Lno])}
1071         end;
1072
1073     ["id", '=', String] when GC#gconf.id == undefined;
1074         GC#gconf.id == "default" ->
1075         fload(FD, GC#gconf{id=String}, Cs, Lno+1, ?NEXTLINE);
1076     ["id", '=', String] ->
1077         error_logger:format("Ignoring 'id = ~p' setting at line ~p~n",
1078             [String,Lno]),
1079         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1080
1081     ["pick_first_virhost_on_nomatch", '=', Bool] ->
1082         case is_bool(Bool) of
1083             {true, Val} ->
1084                 fload(FD, ?gc_set_pick_first_virhost_on_nomatch(GC,Val),
1085                     Cs, Lno+1, ?NEXTLINE);
1086             false ->
1087                 {error, ?F("Expect true|false at line ~w", [Lno])}
1088         end;
1089
1090     ["use_fdsrv", '=', _Bool] ->
1091         %% feature removed
1092         error_logger:format(
1093             "use_fdsrv in yaws.conf is no longer supported - ignoring\n",[]
1094         ),
1095         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1096
1097     ["use_old_ssl", '=', _Bool] ->
1098         %% feature removed
1099         error_logger:format(
1100             "use_old_ssl in yaws.conf is no longer supported - ignoring\n",[]
1101         ),
1102         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1103
1104     ["use_large_ssl_pool", '=', _Bool] ->
1105         %% just ignore - not relevant any longer
1106         error_logger:format(
1107             "use_large_ssl_pool in yaws.conf is no longer supported"
1108             " - ignoring\n", []
1109         ),
1110         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1111
1112     ["x_forwarded_for_log_proxy_whitelist", '=' | _] ->
1113         error_logger:format(
1114             "x_forwarded_for_log_proxy_whitelist in yaws.conf is no longer"
1115             " supported - ignoring\n", []
1116         ),
1117         fload(FD, GC, Cs, Lno+1, ?NEXTLINE);
1118
1119     ["ysession_mod", '=', Mod_str] ->
1120         Ysession_mod = list_to_atom(Mod_str),
1121         fload(FD, GC#gconf{ysession_mod = Ysession_mod}, Cs,
1122             Lno+1, ?NEXTLINE);
1123
1124     ["ysession_cookiegen", '=', Mod_str] ->
1125         Ysession_cookiegen = list_to_atom(Mod_str),
1126         fload(FD, GC#gconf{ysession_cookiegen = Ysession_cookiegen}, Cs,
1127             Lno+1, ?NEXTLINE);
1128
1129     ["ysession_idle_timeout", '=', YsessionIdle] ->
1130         case (catch list_to_integer(YsessionIdle)) of
1131             I when is_integer(I), I > 0 ->
1132                 fload(FD, GC#gconf{ysession_idle_timeout = I}, Cs,
1133                     Lno+1, ?NEXTLINE);
1134             _ ->
1135                 {error, ?F("Expect positive integer at line ~w",[Lno])}
1136         end;
1137
1138     ["ysession_long_timeout", '=', YsessionLong] ->
1139         case (catch list_to_integer(YsessionLong)) of
1140             I when is_integer(I), I > 0 ->
1141                 fload(FD, GC#gconf{ysession_long_timeout = I}, Cs,
1142                     Lno+1, ?NEXTLINE);
1143             _ ->
1144                 {error, ?F("Expect positive integer at line ~w",[Lno])}
1145         end;
1146
1147     ["server_signature", '=', Signature] ->
1148         fload(FD, GC#gconf{yaws=Signature}, Cs, Lno+1, ?NEXTLINE);
1149
1150     ["default_type", '=', MimeType] ->
1151         case parse_mime_types_info(default_type, MimeType,
1152             GC#gconf.mime_types_info,
1153             #mime_types_info{}) of
1154             {ok, Info} ->
1155                 fload(FD, GC#gconf{mime_types_info=Info}, Cs,
1156                     Lno+1, ?NEXTLINE);

```

```

1157         {error, Str} ->
1158             {error, ?F("~s at line ~w", [Str, Lno])}
1159     end;
1160
1161     ["default_charset", '=' | Charset] ->
1162         case parse_mime_types_info(default_charset, Charset,
1163             GC#gconf.mime_types_info,
1164             #mime_types_info{} of
1165             {ok, Info} ->
1166                 fload(FD, GC#gconf{mime_types_info=Info}, Cs,
1167                     Lno+1, ?NEXTLINE);
1168             {error, Str} ->
1169                 {error, ?F("~s at line ~w", [Str, Lno])}
1170         end;
1171
1172     ["mime_types_file", '=' | File] ->
1173         case parse_mime_types_info(mime_types_file, File,
1174             GC#gconf.mime_types_info,
1175             #mime_types_info{} of
1176             {ok, Info} ->
1177                 fload(FD, GC#gconf{mime_types_info=Info}, Cs,
1178                     Lno+1, ?NEXTLINE);
1179             {error, Str} ->
1180                 {error, ?F("~s at line ~w", [Str, Lno])}
1181         end;
1182
1183     ["add_types", '=' | NewTypes] ->
1184         case parse_mime_types_info(add_types, NewTypes,
1185             GC#gconf.mime_types_info,
1186             #mime_types_info{} of
1187             {ok, Info} ->
1188                 fload(FD, GC#gconf{mime_types_info=Info}, Cs,
1189                     Lno+1, ?NEXTLINE);
1190             {error, Str} ->
1191                 {error, ?F("~s at line ~w", [Str, Lno])}
1192         end;
1193
1194     ["add_charsets", '=' | NewCharsets] ->
1195         case parse_mime_types_info(add_charsets, NewCharsets,
1196             GC#gconf.mime_types_info,
1197             #mime_types_info{} of
1198             {ok, Info} ->
1199                 fload(FD, GC#gconf{mime_types_info=Info}, Cs,
1200                     Lno+1, ?NEXTLINE);
1201             {error, Str} ->
1202                 {error, ?F("~s at line ~w", [Str, Lno])}
1203         end;
1204
1205     ["nslookup_pref", '=' | Pref] ->
1206         case parse_nslookup_pref(Pref) of
1207             {ok, Families} ->
1208                 fload(FD, GC#gconf{nslookup_pref = Families}, Cs,
1209                     Lno+1, ?NEXTLINE);
1210             {error, Str} ->
1211                 {error, ?F("~s at line ~w", [Str, Lno])}
1212         end;
1213
1214     ["sni", '=' | Sni] ->
1215         if
1216             Sni == "disable" ->
1217                 fload(FD, GC#gconf{sni=disable}, Cs, Lno+1, ?NEXTLINE);
1218
1219             Sni == "enable" or else Sni == "strict" ->
1220                 case yaws_dynopts:have_ssl_sni() of
1221                     true ->
1222                         fload(FD, GC#gconf{sni=list_to_atom(Sni)}, Cs, Lno+1,
1223                             ?NEXTLINE);
1224                     _ ->
1225                         error_logger:info_msg("Warning, sni option is not"
1226                             " supported at line ~w~n", [Lno]),
1227                         fload(FD, GC, Cs, Lno+1, ?NEXTLINE)
1228                 end;
1229             true ->
1230                 {error, ?F("Expect disable|enable|strict at line ~w",[Lno])}
1231         end;
1232
1233     ['<', "server", Server, '>'] ->
1234         C = #sconf{servername = Server, listen = [],
1235             php_handler = {cgi, GC#gconf.phpexe}},
1236         fload(FD, server, GC, C, Cs, Lno+1, ?NEXTLINE);
1237
1238     [H_] ->
1239         {error, ?F("Unexpected tokens ~p at line ~w", [H, Lno])};
1240     Err ->
1241         Err
1242 end.
1243
1244
1245 fload(FD, server, _GC, _C, _Cs, Lno, eof) ->
1246     file:close(FD),
1247     {error, ?F("Unexpected end-of-file at line ~w", [Lno])};
1248
1249 fload(FD, server, GC, C, Cs, Lno, Chars) ->
1250     case fload(FD, server, GC, C, Lno, Chars) of
1251         {ok, _, _, Lno1, eof} ->
1252             {error, ?F("Unexpected end-of-file at line ~w", [Lno1])};
1253         {ok, GC1, C1, Lno1, ['<', "/server", '>']} ->
1254             HasDocroot =

```

```

1255     case C1#sconf.docroot of
1256     undefined ->
1257         Tests =
1258             [fun() ->
1259                 lists:keymember("/", #proxy_cfg.prefix,
1260                     C1#sconf.revproxy)
1261             end,
1262             fun() ->
1263                 lists:keymember("/", 1,
1264                     C1#sconf.redirect_map)
1265             end,
1266             fun() ->
1267                 lists:foldl(fun(_, true) -> true;
1268                     ({"/", _}, _Acc) -> true;
1269                     (_, Acc) -> Acc
1270                     end, false, C1#sconf.appmods)
1271             end,
1272             fun() ->
1273                 ?sc_forward_proxy(C1)
1274             end],
1275         lists:any(fun(T) -> T() end, Tests);
1276     ->
1277         true
1278     end,
1279     case HasDocroot of
1280     true ->
1281         case C1#sconf.listen of
1282         [] ->
1283             C2 = C1#sconf{listen = {127,0,0,1}},
1284             fload(FD, GC1, [C2|Cs], Lno1+1, ?NEXTLINE);
1285         Ls ->
1286             Cs1 = [C1#sconf{listen=L} || L <- Ls] ++ Cs,
1287             fload(FD, GC1, Cs1, Lno1+1, ?NEXTLINE)
1288         end;
1289     false ->
1290         {error,
1291             ?F("No valid docroot configured for virthost "
1292                 "'~s' (port: ~u)",
1293                 [C1#sconf.servername, C1#sconf.port])}
1294     end;
1295 Err ->
1296     Err
1297 end.
1298
1299 fload(FD, extra_response_headers, GC, C, Lno, Chars) ->
1300     case toks(Lno, Chars) of
1301     [] ->
1302         fload(FD, extra_response_headers, GC, C, Lno+1, ?NEXTLINE);
1303     ["extramod", '=', Mod] ->
1304         ExtraResponseHdrs = C#sconf.extra_response_headers,
1305         C1 = C#sconf{extra_response_headers = [{extramod, list_to_atom(Mod)}|
1306             ExtraResponseHdrs]},
1307         fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1308     ["add", Hdr, '=', Value] ->
1309         ExtraResponseHdrs = C#sconf.extra_response_headers,
1310         C1 = C#sconf{extra_response_headers = [{add,Hdr,Value}|
1311             ExtraResponseHdrs]},
1312         fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1313     ["always", "add", Hdr, '=', Value] ->
1314         ExtraResponseHdrs = C#sconf.extra_response_headers,
1315         C1 = C#sconf{extra_response_headers = [{always_add,Hdr,Value}|
1316             ExtraResponseHdrs]},
1317         fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1318     ["add", Hdr, '='| Value] ->
1319         StringVal = lists:flatten(
1320             yaws:join_sep(
1321                 lists:map(fun(V) when is_atom(V) ->
1322                     atom_to_list(V);
1323                     (V) -> V
1324                     end, Value), " "),
1325             ExtraResponseHdrs = C#sconf.extra_response_headers,
1326             C1 = C#sconf{extra_response_headers = [{add,Hdr,StringVal}|
1327                 ExtraResponseHdrs]},
1328             fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1329     ["always", "add", Hdr, '='| Value] ->
1330         StringVal = lists:flatten(
1331             yaws:join_sep(
1332                 lists:map(fun(V) when is_atom(V) ->
1333                     atom_to_list(V);
1334                     (V) -> V
1335                     end, Value), " "),
1336             ExtraResponseHdrs = C#sconf.extra_response_headers,
1337             C1 = C#sconf{extra_response_headers = [{always_add,Hdr,StringVal}|
1338                 ExtraResponseHdrs]},
1339             fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1340     ["erase", Hdr] ->
1341         ExtraResponseHdrs = C#sconf.extra_response_headers,
1342         C1 = C#sconf{extra_response_headers = [{erase,Hdr}|
1343             ExtraResponseHdrs]},
1344         fload(FD, extra_response_headers, GC, C1, Lno+1, ?NEXTLINE);
1345     ['<', "/extra_response_headers", '>'] ->

```

```

1353         fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
1354
1355     [H|T] ->
1356     {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
1357     Err ->
1358     Err
1359 end;
1360
1361 fload(FD, server, GC, C, Lno, eof) ->
1362     file:close(FD),
1363     {ok, GC, C, Lno, eof};
1364 fload(FD, _, _GC, _C, Lno, eof) ->
1365     file:close(FD),
1366     {error, ?F("Unexpected end-of-file at line ~w", [Lno])};
1367
1368 fload(FD, server, GC, C, Lno, Chars) ->
1369     case toks(Lno, Chars) of
1370     [] ->
1371         fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
1372
1373     ["subconfig", '=', Name] ->
1374         case subconfigfiles(FD, Name, Lno) of
1375         {ok, Files} ->
1376             case fload_subconfigfiles(Files, server, GC, C) of
1377             {ok, GC1, C1} ->
1378                 fload(FD, server, GC1, C1, Lno+1, ?NEXTLINE);
1379             Err ->
1380                 Err
1381             end;
1382             Err ->
1383             Err
1384         end;
1385
1386     ["subconfigdir", '=', Name] ->
1387         case subconfigdir(FD, Name, Lno) of
1388         {ok, Files} ->
1389             case fload_subconfigfiles(Files, server, GC, C) of
1390             {ok, GC1, C1} ->
1391                 fload(FD, server, GC1, C1, Lno+1, ?NEXTLINE);
1392             Err ->
1393                 Err
1394             end;
1395             Err ->
1396             Err
1397         end;
1398
1399     ["server_signature", '=', Sig] ->
1400         fload(FD, server, GC, C#sconf{yaws=Sig}, Lno+1, ?NEXTLINE);
1401
1402     ["access_log", '=', Bool] ->
1403         case is_bool(Bool) of
1404         {true, Val} ->
1405             C1 = ?sc_set_access_log(C, Val),
1406             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1407         false ->
1408             {error, ?F("Expect true|false at line ~w", [Lno])}
1409         end;
1410
1411     ["auth_log", '=', Bool] ->
1412         case is_bool(Bool) of
1413         {true, Val} ->
1414             C1 = ?sc_set_auth_log(C, Val),
1415             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1416         false ->
1417             {error, ?F("Expect true|false at line ~w", [Lno])}
1418         end;
1419
1420     ["logger_mod", '=', Module] ->
1421         C1 = C#sconf{logger_mod = list_to_atom(Module)},
1422         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1423
1424     ["dir_listings", '=', StrVal] ->
1425         case StrVal of
1426         "true" ->
1427             C1 = ?sc_set_dir_listings(C, true),
1428             C2 = ?sc_set_dir_all_zip(C1, true),
1429             C3 = C2#sconf{appmods = [ {"all.zip", yaws_ls},
1430                                     {"all.tgz", yaws_ls},
1431                                     {"all.tbz2", yaws_ls}|
1432                                     C2#sconf.appmods]},
1433             fload(FD, server, GC, C3, Lno+1, ?NEXTLINE);
1434         "true_nozip" ->
1435             C1 = ?sc_set_dir_listings(C, true),
1436             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1437         "false" ->
1438             C1 = ?sc_set_dir_listings(C, false),
1439             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1440         _ ->
1441             {error, ?F("Expect true|true_nozip|false at line ~w",[Lno])}
1442         end;
1443
1444     ["deflate", '=', Bool] ->
1445         case is_bool(Bool) of
1446         {true, Val} ->
1447             C1 = C#sconf{deflate_options=#deflate{}},
1448             C2 = ?sc_set_deflate(C1, Val),
1449             fload(FD, server, GC, C2, Lno+1, ?NEXTLINE);
1450         false ->

```

```

1451         {error, ?F("Expect true|false at line ~w", [Lno])}
1452     end;
1453
1454 ["auth_skip_docroot", '=', Bool] ->
1455     case is_bool(Bool) of
1456     {true, Val} ->
1457         C1 = ?sc_set_auth_skip_docroot(C, Val),
1458         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1459     false ->
1460         {error, ?F("Expect true|false at line ~w", [Lno])}
1461     end;
1462
1463 ["dav", '=', Bool] ->
1464     case is_bool(Bool) of
1465     {true, true} ->
1466         %% Ever since WebDAV support was moved into an appmod,
1467         %% we must no longer set the dav flag in the
1468         %% sconf. Always turn it off instead.
1469         C1 = ?sc_set_dav(C, false),
1470         Runmods = GC#sconf.runmods,
1471         GC1 = case lists:member(yaws_runmod_lock, Runmods) of
1472             false ->
1473                 GC#sconf{runmods=[yaws_runmod_lock|Runmods]};
1474             true ->
1475                 GC
1476         end,
1477         DavAppmods = lists:keystore(yaws_appmod_dav, 2,
1478                                     C1#sconf.appmods,
1479                                     {"/", yaws_appmod_dav}),
1480         C2 = C1#sconf{appmods=DavAppmods},
1481         fload(FD, server, GC1, C2, Lno+1, ?NEXTLINE);
1482     {true, false} ->
1483         C1 = ?sc_set_dav(C, false),
1484         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1485     false ->
1486         {error, ?F("Expect true|false at line ~w", [Lno])}
1487     end;
1488
1489 ["port", '=', Val] ->
1490     case (catch list_to_integer(Val)) of
1491     I when is_integer(I) ->
1492         C1 = C#sconf{port = I},
1493         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1494     _ ->
1495         {error, ?F("Expect integer at line ~w", [Lno])}
1496     end;
1497
1498 ["rmethod", '=', Val] ->
1499     case Val of
1500     "http" ->
1501         C1 = C#sconf{rmethod = Val},
1502         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1503     "https" ->
1504         C1 = C#sconf{rmethod = Val},
1505         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1506     _ ->
1507         {error, ?F("Expect http or https at line ~w", [Lno])}
1508     end;
1509
1510 ["rhost", '=', Val] ->
1511     C1 = C#sconf{rhost = Val},
1512     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1513
1514 ["listen", '=', IP] ->
1515     case inet_parse:address(IP) of
1516     {error, _} ->
1517         {error, ?F("Expect IP address at line ~w:", [Lno])};
1518     {ok, Addr} ->
1519         Lstn = C#sconf.listen,
1520         C1 = if
1521             is_list(Lstn) ->
1522                 case lists:member(Addr, Lstn) of
1523                 false ->
1524                     C#sconf{listen = [Addr|Lstn]};
1525                 true ->
1526                     C
1527             end;
1528             true ->
1529                 C#sconf{listen = [Addr, Lstn]}
1530         end,
1531         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE)
1532     end;
1533
1534 ["listen_backlog", '=', Val] ->
1535     case (catch list_to_integer(Val)) of
1536     B when is_integer(B) ->
1537         C1 = update_options(C, listen_opts, backlog, B),
1538         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1539     _ ->
1540         {error, ?F("Expect integer at line ~w", [Lno])}
1541     end;
1542
1543 ["servername", '=', Name] ->
1544     C1 = ?sc_set_add_port((C#sconf{servername = Name}), false),
1545     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1546
1547 ["serveralias", '=' | Names] ->
1548     C1 = C#sconf{serveralias = Names ++ C#sconf.serveralias},

```



```

1549     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1550
1551 [ '<', "listen_opts", '>' ] ->
1552     fload(FD, listen_opts, GC, C, Lno+1, ?NEXTLINE);
1553
1554 ["docroot", '=' | Rootdir | XtraDirs] ->
1555     RootDirs = lists:map(fun(R) -> filename:absname(R) end,
1556         [Rootdir | XtraDirs]),
1557     case lists:filter(fun(R) -> not is_dir(R) end, RootDirs) of
1558         [] when C#sconf.docroot == undefined ->
1559             C1 = C#sconf{docroot = hd(RootDirs),
1560                 xtra_docroots = tl(RootDirs)},
1561             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1562         [] ->
1563             XtraDocroots = RootDirs ++ C#sconf.xtra_docroots,
1564             C1 = C#sconf{xtra_docroots = XtraDocroots},
1565             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1566         NoDirs ->
1567             error_logger:info_msg("Warning, Skip invalid docroots"
1568                 " at line ~w : ~s~n",
1569                 [Lno, string:join(NoDirs, ", ")]),
1570             case lists:subtract(RootDirs, NoDirs) of
1571                 [] ->
1572                     fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
1573                 [H|T] when C#sconf.docroot == undefined ->
1574                     C1 = C#sconf{docroot = H, xtra_docroots = T},
1575                     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1576                 Ds ->
1577                     XtraDocroots = Ds ++ C#sconf.xtra_docroots,
1578                     C1 = C#sconf{xtra_docroots = XtraDocroots},
1579                     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE)
1580             end
1581     end;
1582
1583 ["partial_post_size", '=', Size] ->
1584     case Size of
1585         "nolimit" ->
1586             C1 = C#sconf{partial_post_size = nolimit},
1587             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1588         Val ->
1589             case (catch list_to_integer(Val)) of
1590                 I when is_integer(I) ->
1591                     C1 = C#sconf{partial_post_size = I},
1592                     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1593                 _ ->
1594                     {error,
1595                     ?F("Expect integer or 'nolimit' at line ~w",
1596                         [Lno])}
1597             end
1598     end;
1599
1600 [ '<', "auth", '>' ] ->
1601     C1 = C#sconf{authdirs=[auth{}]|C#sconf.authdirs},
1602     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
1603
1604 [ '<', "redirect", '>' ] ->
1605     fload(FD, server_redirect, GC, C, Lno+1, ?NEXTLINE);
1606
1607 [ '<', "deflate", '>' ] ->
1608     C1 = C#sconf{deflate_options=#deflate(mime_types=[])},
1609     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
1610
1611 ["default_server_on_this_ip", '=', _Bool] ->
1612     error_logger:format(
1613         "default_server_on_this_ip in yaws.conf is no longer"
1614         " supported - ignoring~n", []
1615     ),
1616     fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
1617
1618 [ '<', "ssl", '>' ] ->
1619     ssl_start(),
1620     fload(FD, ssl, GC, C#sconf{ssl = #ssl{}}, Lno+1, ?NEXTLINE);
1621
1622 ["appmods", '=' | AppMods] ->
1623     case parse_appmods(AppMods, []) of
1624         {ok, L} ->
1625             C1 = C#sconf{appmods = L ++ C#sconf.appmods},
1626             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1627         {error, Str} ->
1628             {error, ?F("~s at line ~w", [Str, Lno])}
1629     end;
1630
1631 ["dispatchmod", '=', DispatchMod] ->
1632     C1 = C#sconf{dispatch_mod = list_to_atom(DispatchMod)},
1633     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1634
1635 ["expires", '=' | Expires] ->
1636     case parse_expires(Expires, []) of
1637         {ok, L} ->
1638             C1 = C#sconf{expires = L ++ C#sconf.expires},
1639             fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1640         {error, Str} ->
1641             {error, ?F("~s at line ~w", [Str, Lno])}
1642     end;
1643
1644 ["errormod_404", '=', Module] ->
1645     C1 = C#sconf{errormod_404 = list_to_atom(Module)},
1646     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);

```

```

1647
1648 ["errormod_crash", '=', Module] ->
1649     C1 = C#sconf(errormod_crash = list_to_atom(Module)),
1650     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1651
1652 ["errormod_401", '=', Module] ->
1653     C1 = C#sconf(errormod_401 = list_to_atom(Module)),
1654     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1655
1656 ["arg_rewrite_mod", '=', Module] ->
1657     C1 = C#sconf(arg_rewrite_mod = list_to_atom(Module)),
1658     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1659
1660 ["tilde_expand", '=', Bool] ->
1661     case is_bool(Bool) of
1662     {true, Val} ->
1663         C1 = ?sc_set_tilde_expand(C, Val),
1664         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1665     false ->
1666         {error, ?F("Expect true|false at line ~w", [Lno])}
1667     end;
1668
1669 ['<', "opaque", '>'] ->
1670     fload(FD, opaque, GC, C, Lno+1, ?NEXTLINE);
1671
1672 ["start_mod", '=', Module] ->
1673     C1 = C#sconf(start_mod = list_to_atom(Module)),
1674     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1675
1676 ['<', "rss", '>'] ->
1677     erase(rss_id),
1678     put(rss, []),
1679     fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
1680
1681 ["tilde_allowed_scripts", '=' | Suffixes] ->
1682     C1 = C#sconf(tilde_allowed_scripts=Suffixes),
1683     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1684
1685 ["allowed_scripts", '=' | Suffixes] ->
1686     C1 = C#sconf(allowed_scripts=Suffixes),
1687     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1688
1689 ["index_files", '=' | Files] ->
1690     case parse_index_files(Files) of
1691     ok ->
1692         C1 = C#sconf(index_files = Files),
1693         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1694     {error, Str} ->
1695         {error, ?F("~s at line ~w", [Str, Lno])}
1696     end;
1697
1698 ["revproxy", '=' | Tail] ->
1699     case parse_revproxy(Tail) of
1700     {ok, RevProxy} ->
1701         C1 = C#sconf{revproxy = [RevProxy | C#sconf.revproxy]},
1702         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1703     {error, url} ->
1704         {error, ?F("Bad url at line ~p", [Lno])};
1705     {error, syntax} ->
1706         {error, ?F("Bad revproxy syntax at line ~p", [Lno])};
1707     Error ->
1708         Error
1709     end;
1710
1711 ["fwdproxy", '=', Bool] ->
1712     case is_bool(Bool) of
1713     {true, Val} ->
1714         C1 = ?sc_set_forward_proxy(C, Val),
1715         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1716     false ->
1717         {error, ?F("Expect true|false at line ~w", [Lno])}
1718     end;
1719
1720 ['<', "extra_cgi_vars", "dir", '=', Dir, '>'] ->
1721     C1 = C#sconf{extra_cgi_vars=[{Dir, []}|C#sconf.extra_cgi_vars]},
1722     fload(FD, extra_cgi_vars, GC, C1, Lno+1, ?NEXTLINE);
1723
1724 ["statistics", '=', Bool] ->
1725     case is_bool(Bool) of
1726     {true, Val} ->
1727         C1 = ?sc_set_statistics(C, Val),
1728         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1729     false ->
1730         {error, ?F("Expect true|false at line ~w", [Lno])}
1731     end;
1732
1733 ["fcgi_app_server", '=' | Val] ->
1734     HostPortSpec = case Val of
1735     [HPS] -> HPS;
1736     ['[', HSpec, ']', PSpec] -> "[" ++ HSpec ++ "]" ++ PSpec
1737     end,
1738     case string_to_host_and_port(HostPortSpec) of
1739     {ok, Host, Port} ->
1740         C1 = C#sconf{fcgi_app_server = {Host, Port}},
1741         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1742     {error, Reason} ->
1743         {error, ?F("Invalid fcgi_app_server ~p at line ~w: ~s",
1744             [HostPortSpec, Lno, Reason])}
1745     end;

```

```

1745     end;
1746
1747 ["fcgi_trace_protocol", '=', Bool] ->
1748     case is_bool(Bool) of
1749     {true, Val} ->
1750         C1 = ?sc_set_fcgi_trace_protocol(C, Val),
1751         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1752     false ->
1753         {error, ?F("Expect true|false at line ~w", [Lno])}
1754     end;
1755
1756 ["fcgi_log_app_error", '=', Bool] ->
1757     case is_bool(Bool) of
1758     {true, Val} ->
1759         C1 = ?sc_set_fcgi_log_app_error(C, Val),
1760         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1761     false ->
1762         {error, ?F("Expect true|false at line ~w", [Lno])}
1763     end;
1764
1765 ["phpfcgi", '=', HostPortSpec] ->
1766     error_logger:format(
1767         "'phpfcgi' is deprecated, use 'php_handler' instead\n", []),
1768     case string_to_host_and_port(HostPortSpec) of
1769     {ok, Host, Port} ->
1770         C1 = C#sconf{php_handler = {fcgi, {Host, Port}}},
1771         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1772     {error, Reason} ->
1773         {error,
1774          ?F("Invalid php fcgi server ~p at line ~w: ~s",
1775            [HostPortSpec, Lno, Reason])}
1776     end;
1777
1778 ["php_handler", '=' | PhpMod] ->
1779     case parse_phpmod(PhpMod, GC#gconf.phpexe) of
1780     {ok, I} ->
1781         C1 = C#sconf{php_handler = I},
1782         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1783     {error, Reason} ->
1784         {error,
1785          ?F("Invalid php_handler configuration at line ~w: ~s",
1786            [Lno, Reason])}
1787     end;
1788
1789 ["shaper", '=', Module] ->
1790     C1 = C#sconf{shaper = list_to_atom(Module)},
1791     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1792
1793
1794 ["default_type", '=', MimeType] ->
1795     case parse_mime_types_info(default_type, MimeType,
1796                                C#sconf.mime_types_info,
1797                                GC#gconf.mime_types_info) of
1798     {ok, Info} ->
1799         fload(FD, server, GC, C#sconf{mime_types_info=Info},
1800             Lno+1, ?NEXTLINE);
1801     {error, Str} ->
1802         {error, ?F("~s at line ~w", [Str, Lno])}
1803     end;
1804
1805 ["default_charset", '=', Charset] ->
1806     case parse_mime_types_info(default_charset, Charset,
1807                                C#sconf.mime_types_info,
1808                                GC#gconf.mime_types_info) of
1809     {ok, Info} ->
1810         fload(FD, server, GC, C#sconf{mime_types_info=Info},
1811             Lno+1, ?NEXTLINE);
1812     {error, Str} ->
1813         {error, ?F("~s at line ~w", [Str, Lno])}
1814     end;
1815
1816 ["mime_types_file", '=', File] ->
1817     case parse_mime_types_info(mime_types_file, File,
1818                                C#sconf.mime_types_info,
1819                                GC#gconf.mime_types_info) of
1820     {ok, Info} ->
1821         fload(FD, server, GC, C#sconf{mime_types_info=Info},
1822             Lno+1, ?NEXTLINE);
1823     {error, Str} ->
1824         {error, ?F("~s at line ~w", [Str, Lno])}
1825     end;
1826
1827 ["add_types", '=' | NewTypes] ->
1828     case parse_mime_types_info(add_types, NewTypes,
1829                                C#sconf.mime_types_info,
1830                                GC#gconf.mime_types_info) of
1831     {ok, Info} ->
1832         fload(FD, server, GC, C#sconf{mime_types_info=Info},
1833             Lno+1, ?NEXTLINE);
1834     {error, Str} ->
1835         {error, ?F("~s at line ~w", [Str, Lno])}
1836     end;
1837
1838 ["add_charsets", '=' | NewCharsets] ->
1839     case parse_mime_types_info(add_charsets, NewCharsets,
1840                                C#sconf.mime_types_info,
1841                                GC#gconf.mime_types_info) of
1842     {ok, Info} ->

```

```

1843         fload(FD, server, GC, C#sconf{mime_types_info=Info},
1844             Lno+1, ?NEXTLINE);
1845     {error, Str} ->
1846         {error, ?F("~s at line ~w", [Str, Lno])}
1847     end;
1848
1849     ["strip_undefined_bindings", '=', Bool] ->
1850         case is_bool(Bool) of
1851             {true, Val} ->
1852                 C1 = ?sc_set_strip_undef_bindings(C, Val),
1853                 fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
1854             false ->
1855                 {error, ?F("Expect true|false at line ~w", [Lno])}
1856         end;
1857
1858     ['<', "extra_response_headers", '>'] ->
1859         fload(FD, extra_response_headers, GC, C, Lno+1, ?NEXTLINE);
1860
1861     ['<', "/server", '>'] ->
1862         {ok, GC, C, Lno, ['<', "/server", '>']};
1863
1864     [H|T] ->
1865         {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
1866     Err ->
1867         Err
1868 end;
1869
1870
1871 fload(FD, listen_opts, GC, C, Lno, Chars) ->
1872     case toks(Lno, Chars) of
1873     [] ->
1874         fload(FD, listen_opts, GC, C, Lno+1, ?NEXTLINE);
1875
1876     ["buffer", '=', Int] ->
1877         case (catch list_to_integer(Int)) of
1878             B when is_integer(B) ->
1879                 C1 = update_options(C, listen_opts, buffer, B),
1880                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1881             _ ->
1882                 {error, ?F("Expect integer at line ~w", [Lno])}
1883         end;
1884
1885     ["delay_send", '=', Bool] ->
1886         case is_bool(Bool) of
1887             {true, Val} ->
1888                 C1 = update_options(C, listen_opts, delay_send, Val),
1889                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1890             false ->
1891                 {error, ?F("Expect true|false at line ~w", [Lno])}
1892         end;
1893
1894     ["linger", '=', Val] ->
1895         case (catch list_to_integer(Val)) of
1896             I when is_integer(I) ->
1897                 C1 = update_options(C, listen_opts, linger, {true, I}),
1898                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1899             _ when Val == "false" ->
1900                 C1 = update_options(C, listen_opts, linger, {false, 0}),
1901                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1902             _ ->
1903                 {error, ?F("Expect integer|false at line ~w", [Lno])}
1904         end;
1905
1906     ["nodelay", '=', Bool] ->
1907         case is_bool(Bool) of
1908             {true, Val} ->
1909                 C1 = update_options(C, listen_opts, nodelay, Val),
1910                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1911             false ->
1912                 {error, ?F("Expect true|false at line ~w", [Lno])}
1913         end;
1914
1915     ["priority", '=', Int] ->
1916         case (catch list_to_integer(Int)) of
1917             P when is_integer(P) ->
1918                 C1 = update_options(C, listen_opts, priority, P),
1919                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1920             _ ->
1921                 {error, ?F("Expect integer at line ~w", [Lno])}
1922         end;
1923
1924     ["sndbuf", '=', Int] ->
1925         case (catch list_to_integer(Int)) of
1926             I when is_integer(I) ->
1927                 C1 = update_options(C, listen_opts, sndbuf, I),
1928                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1929             _ ->
1930                 {error, ?F("Expect integer at line ~w", [Lno])}
1931         end;
1932
1933     ["recbuf", '=', Int] ->
1934         case (catch list_to_integer(Int)) of
1935             I when is_integer(I) ->
1936                 C1 = update_options(C, listen_opts, recbuf, I),
1937                 fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1938             _ ->
1939                 {error, ?F("Expect integer at line ~w", [Lno])}
1940         end;

```

```

1941 ["send_timeout", '=', Val] ->
1942
1943     case (catch list_to_integer(Val)) of
1944     I when is_integer(I) ->
1945         C1 = update_options(C, listen_opts, send_timeout, I),
1946         fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1947     _ when Val == "infinity" ->
1948         C1 = update_options(C, listen_opts, send_timeout,
1949                             infinity),
1950         fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1951     _ ->
1952         {error, ?F("Expect integer|infinity at line ~w", [Lno])}
1953     end;
1954
1955 ["send_timeout_close", '=', Bool] ->
1956     case is_bool(Bool) of
1957     {true, Val} ->
1958         C1 = update_options(C, listen_opts, send_timeout_close,
1959                             Val),
1960         fload(FD, listen_opts, GC, C1, Lno+1, ?NEXTLINE);
1961     false ->
1962         {error, ?F("Expect true|false at line ~w", [Lno])}
1963     end;
1964
1965 ['<', "/listen_opts", '>'] ->
1966     fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
1967
1968 [H|T] ->
1969     {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
1970 Err ->
1971     Err
1972 end;
1973
1974 fload(FD, ssl, GC, C, Lno, Chars) ->
1975     case toks(Lno, Chars) of
1976     [] ->
1977         fload(FD, ssl, GC, C, Lno+1, ?NEXTLINE);
1978
1979     %% A bunch of ssl options
1980
1981     ["keyfile", '=', Val] ->
1982         case is_file(Val) of
1983         true ->
1984             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{keyfile = Val}},
1985             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
1986         _ ->
1987             {error, ?F("Expect existing file at line ~w", [Lno])}
1988         end;
1989
1990     ["certfile", '=', Val] ->
1991         case is_file(Val) of
1992         true ->
1993             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{certfile = Val}},
1994             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
1995         _ ->
1996             {error, ?F("Expect existing file at line ~w", [Lno])}
1997         end;
1998
1999     ["cacertfile", '=', Val] ->
2000         case is_file(Val) of
2001         true ->
2002             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{cacertfile = Val}},
2003             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2004         _ ->
2005             {error, ?F("Expect existing file at line ~w", [Lno])}
2006         end;
2007
2008     ["dhfile", '=', Val] ->
2009         case is_file(Val) of
2010         true ->
2011             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{dhfile = Val}},
2012             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2013         _ ->
2014             {error, ?F("Expect existing file at line ~w", [Lno])}
2015         end;
2016
2017     ["verify", '=', Val0] ->
2018         Fail0 = (C#sconf.ssl)#ssl.fail_if_no_peer_cert,
2019         {Val, Fail} = try
2020             case list_to_integer(Val0) of
2021             0 -> {verify_none, Fail0};
2022             1 -> {verify_peer, false};
2023             2 -> {verify_peer, true};
2024             _ -> {error, Fail0}
2025             end
2026         catch error:badarg ->
2027             case list_to_atom(Val0) of
2028             verify_none -> {verify_none, Fail0};
2029             verify_peer -> {verify_peer, Fail0};
2030             _ -> {error, Fail0}
2031             end
2032         end,
2033     case Val of
2034     error ->
2035         {error, ?F("Expect integer or verify_none, "
2036                   "verify_peer at line ~w", [Lno])};
2037     _ ->
2038         SSL = (C#sconf.ssl)#ssl{verify=Val,

```

```

2039         fail_if_no_peer_cert=Fail},
2040         C1 = C#sconf{ssl=SSL},
2041         fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE)
2042     end;
2043
2044 ["fail_if_no_peer_cert", '=', Bool] ->
2045     case is_bool(Bool) of
2046     {true, Val} ->
2047         C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{
2048             fail_if_no_peer_cert = Val}},
2049         fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2050     false ->
2051         {error, ?F("Expect true|false at line ~w", [Lno])}
2052     end;
2053
2054 ["depth", '=', Val0] ->
2055     Val = (catch list_to_integer(Val0)),
2056     case lists:member(Val, [0, 1,2,3,4,5,6,7]) of
2057     true ->
2058         C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{depth = Val}},
2059         fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2060     _ ->
2061         {error, ?F("Expect integer 0..7 at line ~w", [Lno])}
2062     end;
2063
2064 ["password", '=', Val] ->
2065     C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{password = Val}},
2066     fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2067
2068 ["ciphers", '=', Val] ->
2069     try
2070         L = str2term(Val),
2071         Ciphers = ssl:cipher_suites(),
2072         case check_ciphers(L, Ciphers) of
2073         ok ->
2074             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{ciphers = L}},
2075             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2076         Err ->
2077             Err
2078         end
2079     catch _:_ ->
2080         {error, ?F("Bad cipherspec at line ~w", [Lno])}
2081     end;
2082
2083 ["ecsc", '=', Val] ->
2084     try
2085         L = str2term(Val),
2086         Curves = ssl:ecscs(),
2087         case check_ecsc(L, Curves) of
2088         ok ->
2089             C1 = C#sconf{ssl = (C#sconf.ssl)#ssl{ecsc = L}},
2090             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2091         Err ->
2092             Err
2093         end
2094     catch _:_ ->
2095         {error, ?F("Bad elliptic curves at line ~w", [Lno])}
2096     end;
2097
2098 ["secure_renegotiate", '=', Bool] ->
2099     case is_bool(Bool) of
2100     {true, Val} ->
2101         C1 = C#sconf{ssl=(C#sconf.ssl)#ssl{secure_renegotiate=Val}},
2102         fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2103     false ->
2104         {error, ?F("Expect true|false at line ~w", [Lno])}
2105     end;
2106
2107 ["client_renegotiation", '=', Bool] ->
2108     case yaws_dynopts:have_ssl_client_renegotiation() of
2109     true ->
2110         case is_bool(Bool) of
2111         {true, Val} ->
2112             C1 = C#sconf{ssl=(C#sconf.ssl)#ssl{client_renegotiation=Val}},
2113             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2114         false ->
2115             {error, ?F("Expect true|false at line ~w", [Lno])}
2116         end;
2117     _ ->
2118         error_logger:info_msg("Warning, client_renegotiation SSL "
2119             "option is not supported "
2120             "at line ~w~n", [Lno]),
2121         fload(FD, ssl, GC, C, Lno+1, ?NEXTLINE)
2122     end;
2123
2124 ["honor_cipher_order", '=', Bool] ->
2125     case yaws_dynopts:have_ssl_honor_cipher_order() of
2126     true ->
2127         case is_bool(Bool) of
2128         {true, Val} ->
2129             C2 = C#sconf{
2130                 ssl=(C#sconf.ssl)#ssl{honor_cipher_order=Val}
2131             },
2132             fload(FD, ssl, GC, C2, Lno+1, ?NEXTLINE);
2133         false ->
2134             {error, ?F("Expect true|false at line ~w", [Lno])}
2135         end;
2136     _ ->
2137         error_logger:info_msg("Warning, honor_cipher_order SSL "
2138             "option is not supported "

```

```

2137         "at line ~w~n", [Lno]),
2138         fload(FD, ssl, GC, C, Lno+1, ?NEXTLINE)
2139     end;
2140
2141     ["protocol_version", '=' | Vsns0] ->
2142     try
2143         Vsns = [list_to_existing_atom(V) || V <- Vsns0, not is_atom(V)],
2144         C1 = C#sconf{
2145             ssl=(C#sconf.ssl)#ssl{protocol_version=Vsns}
2146         },
2147         fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE)
2148     catch _:_ ->
2149         {error, ?F("Bad ssl protocol_version at line ~w", [Lno])}
2150     end;
2151
2152     ["require_sni", '=', Bool] ->
2153     case is_bool(Bool) of
2154         {true, Val} ->
2155             C1 = C#sconf{
2156                 ssl=(C#sconf.ssl)#ssl{require_sni=Val}
2157             },
2158             fload(FD, ssl, GC, C1, Lno+1, ?NEXTLINE);
2159         false ->
2160             {error, ?F("Expect true|false at line ~w", [Lno])}
2161     end;
2162
2163     ['<', "/ssl", '>'] ->
2164         fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
2165
2166     [H|T] ->
2167         {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2168     Err ->
2169         Err
2170 end;
2171
2172 fload(FD, server_auth, GC, C, Lno, Chars) ->
2173     [Auth|AuthDirs] = C#sconf.authdirs,
2174     case toks(Lno, Chars) of
2175         [] ->
2176             fload(FD, server_auth, GC, C, Lno+1, ?NEXTLINE);
2177
2178         ["docroot", '=', Docroot] ->
2179             Auth1 = Auth#auth{docroot = filename:absname(Docroot)},
2180             C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2181             fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2182
2183         ["dir", '=', Dir] ->
2184             case file:list_dir(Dir) of
2185                 {ok, _} when Dir /= "/" ->
2186                     error_logger:info_msg("Warning, authdir must be set "
2187                         "relative docroot ~n", []);
2188                 _ ->
2189                     ok
2190             end,
2191             Dir1 = yaws_api:path_norm(Dir),
2192             Auth1 = Auth#auth{dir = [Dir1 | Auth#auth.dir]},
2193             C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2194             fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2195
2196         ["realm", '=', Realm] ->
2197             Auth1 = Auth#auth{realm = Realm},
2198             C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2199             fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2200
2201         ["authmod", '=', Mod] ->
2202             Mod1 = list_to_atom(Mod),
2203             code:ensure_loaded(Mod1),
2204             %% Add the auth header for the mod
2205             H = try
2206                 Mod1:get_header() ++ Auth#auth.headers
2207             catch _:_ ->
2208                 error_logger:format("Failed to ~p:get_header() ~n",
2209                     [Mod1]),
2210                 Auth#auth.headers
2211             end,
2212             Auth1 = Auth#auth{mod = Mod1, headers = H},
2213             C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2214             fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2215
2216         ["user", '=', User] ->
2217             case parse_auth_user(User, Lno) of
2218                 {Name, Algo, Salt, Hash} ->
2219                     Auth1 = Auth#auth{
2220                         users = [{Name, Algo, Salt, Hash}|Auth#auth.users]
2221                     },
2222                     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2223                     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2224                 {error, Str} ->
2225                     {error, Str}
2226             end;
2227
2228         ["allow", '=', "all"] ->
2229             Auth1 = case Auth#auth.acl of
2230                 none -> Auth#auth{acl={all, [], deny_allow}};
2231                 {_, D, 0} -> Auth#auth{acl={all, D, 0}}
2232             end,
2233             C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2234             fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);

```

```

2235
2236 ["allow", '=' | IPs] ->
2237     Auth1 = case Auth#auth.acl of
2238         none ->
2239             AllowIPs = parse_auth_ips(IPs, []),
2240             Auth#auth{acl={AllowIPs, [], deny_allow}};
2241         {all, _, _} ->
2242             Auth;
2243         {AllowIPs, DenyIPs, Order} ->
2244             AllowIPs1 = parse_auth_ips(IPs, []) ++ AllowIPs,
2245             Auth#auth{acl={AllowIPs1, DenyIPs, Order}}
2246     end,
2247     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2248     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2249
2250 ["deny", '=', "all"] ->
2251     Auth1 = case Auth#auth.acl of
2252         none -> Auth#auth{acl={[], all, deny_allow}};
2253         {A, _, 0} -> Auth#auth{acl={A, all, 0}}
2254     end,
2255     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2256     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2257
2258 ["deny", '=' | IPs] ->
2259     Auth1 = case Auth#auth.acl of
2260         none ->
2261             DenyIPs = parse_auth_ips(IPs, []),
2262             Auth#auth{acl={[], DenyIPs, deny_allow}};
2263         {_, all, _} ->
2264             Auth;
2265         {AllowIPs, DenyIPs, Order} ->
2266             DenyIPs1 = parse_auth_ips(IPs, []) ++ DenyIPs,
2267             Auth#auth{acl={AllowIPs, DenyIPs1, Order}}
2268     end,
2269     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2270     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2271
2272 ["order", '=', "allow", ',', "deny"] ->
2273     Auth1 = case Auth#auth.acl of
2274         none -> Auth#auth{acl={[], [], allow_deny}};
2275         {A,D,_} -> Auth#auth{acl={A, D, allow_deny}}
2276     end,
2277     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2278     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2279
2280 ["order", '=', "deny", ',', "allow"] ->
2281     Auth1 = case Auth#auth.acl of
2282         none -> Auth#auth{acl={[], [], deny_allow}};
2283         {A,D,_} -> Auth#auth{acl={A, D, deny_allow}}
2284     end,
2285     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2286     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2287
2288 ["pam", "service", '=', Serv] ->
2289     Auth1 = Auth#auth{pam=Serv},
2290     C1 = C#sconf{authdirs=[Auth1|AuthDirs]},
2291     fload(FD, server_auth, GC, C1, Lno+1, ?NEXTLINE);
2292
2293 ['<', "/auth", '>'] ->
2294     Pam = Auth#auth.pam,
2295     Users = Auth#auth.users,
2296     Realm = Auth#auth.realm,
2297     Auth1 = case {Pam, Users} of
2298         {false, []} ->
2299             Auth;
2300         _ ->
2301             H = Auth#auth.headers ++
2302                 yaws:make_www_authenticate_header({realm, Realm}),
2303             Auth#auth{headers = H}
2304     end,
2305     AuthDirs1 = case Auth1#auth.dir of
2306         [] -> [Auth1#auth{dir="/" }|AuthDirs];
2307         Ds -> [Auth1#auth{dir=D} || D <- Ds] ++ AuthDirs
2308     end,
2309     C1 = C#sconf{authdirs=AuthDirs1},
2310     fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
2311
2312 [H|T] ->
2313     {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2314 Err ->
2315     Err
2316 end;
2317
2318 fload(FD, server_redirect, GC, C, Lno, Chars) ->
2319     RedirMap = C#sconf.redirect_map,
2320     case toks(Lno, Chars) of
2321     [] ->
2322         fload(FD, server_redirect, GC, C, Lno+1, ?NEXTLINE);
2323
2324     [Path, '=', '=' | Rest] ->
2325         %% "Normalize" Path
2326         Path1 = filename:join([yaws_api:path_norm(Path)]),
2327         case parse_redirect(Path1, Rest, noappend, Lno) of
2328         {error, Str} ->
2329             {error, Str};
2330         Redir ->
2331             C1 = C#sconf{redirect_map=RedirMap ++ [Redir]},
2332             fload(FD, server_redirect, GC, C1, Lno+1, ?NEXTLINE)
2333
2334

```



```

2333     end;
2334
2335     [Path, '=' | Rest] ->
2336         %% "Normalize" Path
2337         Path1 = filename:join([yaws_api:path_norm(Path)]),
2338         case parse_redirect(Path1, Rest, append, Lno) of
2339             {error, Str} ->
2340                 {error, Str};
2341             Redir ->
2342                 C1 = C$conf{redirect_map=RedirMap ++ [Redir]},
2343                 fload(FD, server_redirect, GC, C1, Lno+1, ?NEXTLINE)
2344         end;
2345
2346     ['<', "/redirect", '>'] ->
2347         fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
2348
2349     [H|T] ->
2350         {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2351     Err ->
2352         Err
2353 end;
2354
2355 fload(FD, server_deflate, GC, C, Lno, Chars) ->
2356     Deflate = C$conf.deflate_options,
2357     case toks(Lno, Chars) of
2358         [] ->
2359             fload(FD, server_deflate, GC, C, Lno+1, ?NEXTLINE);
2360
2361         ["min_compress_size", '=', CSize] ->
2362             case (catch list_to_integer(CSize)) of
2363                 I when is_integer(I), I > 0 ->
2364                     Deflate1 = Deflate#deflate{min_compress_size=I},
2365                     C1 = C$conf{deflate_options=Deflate1},
2366                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2367                 _ when CSize == "nolimit" ->
2368                     Deflate1 = Deflate#deflate{min_compress_size=nolimit},
2369                     C1 = C$conf{deflate_options=Deflate1},
2370                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2371                 _ ->
2372                     {error, ?F("Expect integer > 0 at line ~w", [Lno])}
2373             end;
2374
2375         ["mime_types", '=' | MimeTypes] ->
2376             case parse_compressible_mime_types(MimeTypes,
2377                 Deflate#deflate.mime_types) of
2378                 {ok, L} ->
2379                     Deflate1 = Deflate#deflate{mime_types=L},
2380                     C1 = C$conf{deflate_options=Deflate1},
2381                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2382                 {error, Str} ->
2383                     {error, ?F("~s at line ~w", [Str, Lno])}
2384             end;
2385
2386         ["compression_level", '=', CLevel] ->
2387             L = try
2388                 list_to_integer(CLevel)
2389             catch error:badarg ->
2390                 list_to_atom(CLevel)
2391             end,
2392             if
2393                 L ==: none; L ==: default;
2394                 L ==: best_compression; L ==: best_speed ->
2395                     Deflate1 = Deflate#deflate{compression_level=L},
2396                     C1 = C$conf{deflate_options=Deflate1},
2397                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2398                 is_integer(L), L >= 0, L <= 9 ->
2399                     Deflate1 = Deflate#deflate{compression_level=L},
2400                     C1 = C$conf{deflate_options=Deflate1},
2401                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2402                 true ->
2403                     {error, ?F("Bad compression level at line ~w", [Lno])}
2404             end;
2405
2406         ["window_size", '=', WSize] ->
2407             case (catch list_to_integer(WSize)) of
2408                 I when is_integer(I), I > 8, I < 16 ->
2409                     Deflate1 = Deflate#deflate{window_size=I * -1},
2410                     C1 = C$conf{deflate_options=Deflate1},
2411                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2412                 _ ->
2413                     {error,
2414                         ?F("Expect integer between 9..15 at line ~w",
2415                             [Lno])}
2416             end;
2417
2418         ["mem_level", '=', MLevel] ->
2419             case (catch list_to_integer(MLevel)) of
2420                 I when is_integer(I), I >= 1, I <= 9 ->
2421                     Deflate1 = Deflate#deflate{mem_level=I},
2422                     C1 = C$conf{deflate_options=Deflate1},
2423                     fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2424                 _ ->
2425                     {error, ?F("Expect integer between 1..9 at line ~w", [Lno])}
2426             end;
2427
2428         ["strategy", '=', Strategy] ->
2429             if
2430                 Strategy ==: "default";

```

```

2431     Strategy := "filtered";
2432     Strategy := "huffman_only" ->
2433         Deflate1 = Deflate#deflate(strategy=list_to_atom(Strategy)),
2434         C1 = C#sconf{deflate_options=Deflate1},
2435         fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2436     true ->
2437         {error,
2438             ?F("Unknown strategy ~p at line ~w", [Strategy, Lno])}
2439     end;
2440
2441     ["use_gzip_static", '=', Bool] ->
2442         case is_bool(Bool) of
2443             {true, Val} ->
2444                 Deflate1 = Deflate#deflate{use_gzip_static=Val},
2445                 C1 = C#sconf{deflate_options=Deflate1},
2446                 fload(FD, server_deflate, GC, C1, Lno+1, ?NEXTLINE);
2447             false ->
2448                 {error, ?F("Expect true|false at line ~w", [Lno])}
2449         end;
2450
2451     ['<', "/deflate", '>'] ->
2452         Deflate1 = case Deflate#deflate.mime_types of
2453             [] ->
2454                 Deflate#deflate{
2455                     mime_types = ?DEFAULT_COMPRESSIBLE_MIME_TYPES
2456                 };
2457             _ ->
2458                 Deflate
2459         end,
2460         C1 = C#sconf{deflate_options = Deflate1},
2461         fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
2462
2463     [H|T] ->
2464         {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2465     Err ->
2466         Err
2467     end;
2468
2469     fload(FD, extra_cgi_vars, GC, C, Lno, Chars) ->
2470         [{Dir, Vars}|EVars] = C#sconf.extra_cgi_vars,
2471         case toks(Lno, Chars) of
2472             [] ->
2473                 fload(FD, extra_cgi_vars, GC, C, Lno+1, ?NEXTLINE);
2474
2475             [Var, '=', Val] ->
2476                 C1 = C#sconf{extra_cgi_vars=[{Dir, [{Var, Val} | Vars]}|EVars]},
2477                 fload(FD, extra_cgi_vars, GC, C1, Lno+1, ?NEXTLINE);
2478
2479             ['<', "/extra_cgi_vars", '>'] ->
2480                 C1 = C#sconf{extra_cgi_vars = [EVars | C#sconf.extra_cgi_vars]},
2481                 fload(FD, server, GC, C1, Lno+1, ?NEXTLINE);
2482
2483             [H|T] ->
2484                 {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2485             Err ->
2486                 Err
2487         end;
2488
2489     fload(FD, rss, GC, C, Lno, Chars) ->
2490         case toks(Lno, Chars) of
2491             [] ->
2492                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2493
2494             ["rss_id", '=', Value] -> % mandatory !!
2495                 put(rss_id, list_to_atom(Value)),
2496                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2497
2498             ["rss_dir", '=', Value] -> % mandatory !!
2499                 put(rss, [{db_dir, Value} | get(rss)]),
2500                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2501
2502             ["rss_expire", '=', Value] ->
2503                 put(rss, [{expire, Value} | get(rss)]),
2504                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2505
2506             ["rss_days", '=', Value] ->
2507                 put(rss, [{days, Value} | get(rss)]),
2508                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2509
2510             ["rss_rm_exp", '=', Value] ->
2511                 put(rss, [{rm_exp, Value} | get(rss)]),
2512                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2513
2514             ["rss_max", '=', Value] ->
2515                 put(rss, [{rm_max, Value} | get(rss)]),
2516                 fload(FD, rss, GC, C, Lno+1, ?NEXTLINE);
2517
2518             ['<', "/rss", '>'] ->
2519                 case get(rss_id) of
2520                     undefined ->
2521                         {error, ?F("No rss_id specified at line ~w", [Lno])};
2522                     RSSId ->
2523                         yaws_rss:open(RSSId, get(rss)),
2524                         fload(FD, server, GC, C, Lno+1, ?NEXTLINE)
2525                 end;
2526
2527             [H|T] ->
2528                 {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};

```

```

2529     Err ->
2530     Err
2531 end;
2532
2533 fload(FD, opaque, GC, C, Lno, Chars) ->
2534   case toks(Lno, Chars) of
2535   [] ->
2536     fload(FD, opaque, GC, C, Lno+1, ?NEXTLINE);
2537
2538   [Key, '=', Value] ->
2539     C1 = C#sconf{opaque = [{Key, Value} | C#sconf.opaque]},
2540     fload(FD, opaque, GC, C1, Lno+1, ?NEXTLINE);
2541
2542   [Key, '=' | Value] ->
2543     String_value = lists:flatten(
2544       lists:map(
2545         fun(Item) when is_atom(Item) ->
2546           atom_to_list(Item);
2547         (Item) ->
2548           Item
2549       end, Value)),
2550     C1 = C#sconf{opaque = [{Key, String_value} | C#sconf.opaque]},
2551     fload(FD, opaque, GC, C1, Lno+1, ?NEXTLINE);
2552
2553   ['<', "/opaque", '>'] ->
2554     fload(FD, server, GC, C, Lno+1, ?NEXTLINE);
2555
2556   [H|T] ->
2557     {error, ?F("Unexpected input ~p at line ~w", [[H|T], Lno])};
2558   Err ->
2559   Err
2560 end.
2561
2562 is_bool("true") ->
2563   {true, true};
2564 is_bool("false") ->
2565   {true, false};
2566 is_bool(_) ->
2567   false.
2568
2569
2570 warn_dir(Type, Dir) ->
2571   case is_dir(Dir) of
2572   true ->
2573     true;
2574   false ->
2575     error_logger:format("Config Warning: Directory ~s "
2576       "for ~s doesn't exist~n",
2577       [Dir, Type]),
2578     false
2579   end.
2580
2581 is_dir(Val) ->
2582   case file:read_file_info(Val) of
2583   {ok, FI} when FI#file_info.type == directory ->
2584     true;
2585   _ ->
2586     false
2587   end.
2588
2589
2590 is_file(Val) ->
2591   case file:read_file_info(Val) of
2592   {ok, FI} when FI#file_info.type == regular ->
2593     true;
2594   _ ->
2595     false
2596   end.
2597
2598 is_wildcard(Val) ->
2599   (lists:member($*, Val) orelse
2600    lists:member($?, Val) orelse
2601    (lists:member($[, Val] andalso lists:member($, Val)) orelse
2602     (lists:member($[, Val] andalso lists:member($, Val)))).
2603
2604
2605 %% tokenizer
2606 toks(Lno, Chars) ->
2607   toks(Lno, Chars, free, [], []). % two accumulators
2608
2609 toks(Lno, [$_|_T], Mode, Ack, Tack) ->
2610   toks(Lno, [], Mode, Ack, Tack);
2611
2612 toks(Lno, [H|T], free, Ack, Tack) ->
2613   %%?Debug("Char=~p", [H]),
2614   case {is_quote(H), is_string_char([H|T], is_special(H), yaws:is_space(H))} of
2615   {_, _} -> true ->
2616     toks(Lno, T, free, Ack, Tack);
2617   {_, _}, true, _ ->
2618     toks(Lno, T, free, [], [list_to_atom([H]) | Tack]);
2619   {_, true, _} ->
2620     toks(Lno, T, string, [H], Tack);
2621   {_, utf8, _} ->
2622     toks(Lno, tl(T), string, [H, hd(T)], Tack);
2623   {true, _, _} ->
2624     toks(Lno, T, quote, [], Tack);
2625   {false, false, false, false} ->
2626     {error, ?F("Unexpected character <~p / ~c> at line ~w",

```

```

2627         [H,H, Lno]))
2628     end;
2629     toks(Lno, [C|T], string, Ack, Tack) ->
2630     case {is_backquote(C), is_quote(C), is_string_char([C|T]), is_special(C),
2631           yaws:is_space(C)} of
2632     {true, _, _, _} ->
2633         toks(Lno, T, [backquote,string], Ack, Tack);
2634     {_, _, true, _} ->
2635         toks(Lno, T, string, [C|Ack], Tack);
2636     {_, _, utf8, _} ->
2637         toks(Lno, tl(T), string, [C, hd(T)|Ack], Tack);
2638     {_, _, _, true, _} ->
2639         toks(Lno, T, free, [], [list_to_atom([C]),lists:reverse(Ack)|Tack]);
2640     {_, true, _, _, _} ->
2641         toks(Lno, T, quote, [], [lists:reverse(Ack)|Tack]);
2642     {_, _, _, true} ->
2643         toks(Lno, T, free, [], [lists:reverse(Ack)|Tack]);
2644     {false, false, false, false, false} ->
2645         {error, ?F("Unexpected character <~p / ~c> at line ~w",
2646                   [C, C, Lno]))}
2647     end;
2648     toks(Lno, [C|T], quote, Ack, Tack) ->
2649     case {is_quote(C), is_backquote(C)} of
2650     {true, _} ->
2651         toks(Lno, T, free, [], [lists:reverse(Ack)|Tack]);
2652     {_, true} ->
2653         toks(Lno, T, [backquote,quote], [C|Ack], Tack);
2654     {false, false} ->
2655         toks(Lno, T, quote, [C|Ack], Tack)
2656     end;
2657     toks(Lno, [C|T], [backquote,Mode], Ack, Tack) ->
2658         toks(Lno, T, Mode, [C|Ack], Tack);
2659     toks(_Lno, [], string, Ack, Tack) ->
2660         lists:reverse([lists:reverse(Ack) | Tack]);
2661     toks(_Lno, [], free, _,Tack) ->
2662         lists:reverse(Tack).
2663
2664     is_quote(34) -> true ; %% $" but emacs mode can't handle it
2665     is_quote(_) -> false.
2666
2667     is_backquote($\\) -> true ;
2668     is_backquote(_) -> false.
2669
2670     is_string_char([C|T]) ->
2671     if
2672         $a =< C, C =< $z ->
2673             true;
2674         $A =< C, C =< $Z ->
2675             true;
2676         $0 =< C, C =< $9 ->
2677             true;
2678         C == 195 , T /= [] ->
2679             %% FIXME check that [C, hd(T)] really is a char ?? how
2680             utf8;
2681         true ->
2682             lists:member(C, [$. , $/, $:, $_, $-, $+, $~, $@, $*, $?])
2683     end.
2684
2685     is_special(C) ->
2686         lists:member(C, [$=, $[, $], $[, $], $, $<, $>, $,]).
2687
2688     %% parse the argument string PLString which can either be the undefined
2689     %% atom or a proplist. Currently the only supported keys are
2690     %% fullsweep_after, min_heap_size, and min_bin_vheap_size. Any other
2691     %% key/values are ignored.
2692     parse_process_options(PLString) ->
2693     case erl_scan:string(PLString ++ ".") of
2694     {ok, PLTokens, _} ->
2695         case erl_parse:parse_term(PLTokens) of
2696         {ok, undefined} ->
2697             {ok, []};
2698         {ok, []} ->
2699             {ok, []};
2700         {ok, [Hd|_Tl]=Plist} when is_atom(Hd); is_tuple(Hd) ->
2701             %% create new safe proplist of desired options
2702             {ok, proplists_int_copy([], Plist, [fullsweep_after,
2703                                                  min_heap_size,
2704                                                  min_bin_vheap_size])};
2705         _ ->
2706             {error, "Expect undefined or proplist"}
2707         end;
2708     _ ->
2709         {error, "Expect undefined or proplist"}
2710     end.
2711
2712     %% copy proplist integer values for the given keys from the
2713     %% Src proplist to the Dest proplist. Ignored keys that are not
2714     %% found or have non-integer values. Returns the new Dest proplist.
2715     proplists_int_copy(Dest, _Src, []) ->
2716         Dest;
2717     proplists_int_copy(Dest, Src, [Key|NextKeys]) ->
2718     case proplists:get_value(Key, Src) of
2719     Val when is_integer(Val) ->
2720         proplists_int_copy([{{Key, Val}}|Dest], Src, NextKeys);
2721     _ ->
2722         proplists_int_copy(Dest, Src, NextKeys)
2723     end.
2724

```

```

2725 parse_soap_srv_mods(['<', Module, ',', Handler, ',', WsdlFile, '>' | Tail],
2726     Ack) ->
2727     case is_file(WsdlFile) of
2728     true ->
2729         S = { {list_to_atom(Module), list_to_atom(Handler)}, WsdlFile},
2730         parse_soap_srv_mods(Tail, [S | Ack]);
2731     false ->
2732         {error, ?F("Bad wsdl file ~p", [WsdlFile])}
2733     end;
2734
2735 parse_soap_srv_mods(['<', Module, ',', Handler, ',', WsdlFile, ',',
2736     Prefix, '>' | Tail], Ack) ->
2737     case is_file(WsdlFile) of
2738     true ->
2739         S = { {list_to_atom(Module), list_to_atom(Handler)},
2740             WsdlFile, Prefix},
2741         parse_soap_srv_mods(Tail, [S | Ack]);
2742     false ->
2743         {error, ?F("Bad wsdl file ~p", [WsdlFile])}
2744     end;
2745
2746 parse_soap_srv_mods([ SoapSrvMod | _Tail], _Ack) ->
2747     {error, ?F("Bad soap_srv_mods syntax: ~p", [SoapSrvMod])};
2748
2749 parse_soap_srv_mods([], Ack) ->
2750     {ok, Ack}.
2751
2752 parse_appmods(['<', PathElem, ',', AppMod, '>' | Tail], Ack) ->
2753     S = {PathElem, list_to_atom(AppMod)},
2754     parse_appmods(Tail, [S | Ack]);
2755
2756 parse_appmods(['<', PathElem, ',', AppMod, "exclude_paths" | Tail], Ack) ->
2757     Paths = lists:takewhile(fun(X) -> X /= '>' end,
2758         Tail),
2759     Tail2 = lists:dropwhile(fun(X) -> X /= '>' end,
2760         Tail),
2761     Tail3 = tl(Tail2),
2762
2763     S = {PathElem, list_to_atom(AppMod), lists:map(
2764         fun(Str) ->
2765             string:tokens(Str, "/")
2766         end, Paths)},
2767     parse_appmods(Tail3, [S | Ack]);
2768
2769
2770 parse_appmods([AppMod | Tail], Ack) ->
2771     %% just some simpleminded test to catch syntax errors in the config
2772     case AppMod of
2773     [Char] ->
2774         case is_special(Char) of
2775         true ->
2776             {error, "Bad appmod syntax"};
2777         false ->
2778             S = {AppMod, list_to_atom(AppMod)},
2779             parse_appmods(Tail, [S | Ack])
2780         end;
2781     _ ->
2782         S = {AppMod, list_to_atom(AppMod)},
2783         parse_appmods(Tail, [S | Ack])
2784     end;
2785
2786 parse_appmods([], Ack) ->
2787     {ok, Ack}.
2788
2789
2790 parse_revproxy([Prefix, Uri]) ->
2791     parse_revproxy_url(Prefix, Uri);
2792
2793 parse_revproxy([Prefix, Uri, "intercept_mod", InterceptMod]) ->
2794     case parse_revproxy_url(Prefix, Uri) of
2795     {ok, RP} ->
2796         {ok, RP#proxy_cfg{intercept_mod = list_to_atom(InterceptMod)}};
2797     Error ->
2798         Error
2799     end;
2800
2801 parse_revproxy([Prefix, Proto, '[', IPv6, ']', Rest, "intercept_mod", InterceptMod]) ->
2802     Uri = Proto ++ "[" ++ IPv6 ++ "]" ++ Rest,
2803     parse_revproxy([Prefix, Uri, "intercept_mod", InterceptMod]);
2804
2805 parse_revproxy([Prefix, Proto, '[', IPv6, ']', Rest]) ->
2806     Uri = Proto ++ "[" ++ IPv6 ++ "]" ++ Rest,
2807     parse_revproxy([Prefix, Uri]);
2808
2809 parse_revproxy(_Other) ->
2810     {error, syntax}.
2811
2812
2813 parse_revproxy_url(Prefix, Uri) ->
2814     case (catch yaws_api:parse_url(Uri)) of
2815     {'EXIT', _} ->
2816         {error, url};
2817     URL when URL#url.path == "/" ->
2818         P = case lists:reverse(Prefix) of
2819             [_ | _Tail] ->
2820                 Prefix;
2821             Other ->
2822                 lists:reverse(Other)
2823         end,
2824         {ok, #proxy_cfg{prefix=P, url=URL}};
2825     _URL ->
2826         {error, "Can't revproxy to a URL with a path "}
2827     end.

```

```

2823
2824
2825 parse_expires(['<', MimeType, ',', Expire, '>' | Tail], Acc) ->
2826 {EType, Value} =
2827     case string:tokens(Expire, "+") of
2828     ["always"] ->
2829         {always, 0};
2830     [Secs] ->
2831         {access, (catch list_to_integer(Secs))};
2832     ["access", Secs] ->
2833         {access, (catch list_to_integer(Secs))};
2834     ["modify", Secs] ->
2835         {modify, (catch list_to_integer(Secs))};
2836     _ ->
2837         {error, "Bad expires syntax"}
2838     end,
2839 if
2840     EType == error ->
2841         {EType, Value};
2842     not is_integer(Value) ->
2843         {error, "Bad expires syntax"};
2844     true ->
2845         case parse_mime_type(MimeType) of
2846         {ok, "*", "*"} ->
2847             E = {all, EType, Value},
2848             parse_expires(Tail, [E | Acc]);
2849         {ok, Type, "*"} ->
2850             E = {{Type, all}, EType, Value},
2851             parse_expires(Tail, [E | Acc]);
2852         {ok, _Type, _SubType} ->
2853             E = {MimeType, EType, Value},
2854             parse_expires(Tail, [E | Acc]);
2855         Error ->
2856             Error
2857         end
2858     end;
2859 parse_expires([], Acc)->
2860 {ok, Acc}.
2861
2862
2863 parse_phpmod(['<', "cgi", ',', DefaultPhpPath, '>'], DefaultPhpPath) ->
2864 {ok, {cgi, DefaultPhpPath}};
2865 parse_phpmod(['<', "cgi", ',', PhpPath, '>'], _) ->
2866     case is_file(PhpPath) of
2867     true ->
2868         {ok, {cgi, PhpPath}};
2869     false ->
2870         {error, ?F("~s is not a regular file", [PhpPath])}
2871     end;
2872 parse_phpmod(['<', "fcgi", ',', HostPortSpec, '>'], _) ->
2873     case string_to_host_and_port(HostPortSpec) of
2874     {ok, Host, Port} ->
2875         {ok, {fcgi, {Host, Port}}};
2876     {error, Reason} ->
2877         {error, Reason}
2878     end;
2879 parse_phpmod(['<', "fcgi", ',', '[' , HostSpec, ']', PortSpec, '>'], _) ->
2880     case string_to_host_and_port "[" ++ HostSpec ++ "]" ++ PortSpec) of
2881     {ok, Host, Port} ->
2882         {ok, {fcgi, {Host, Port}}};
2883     {error, Reason} ->
2884         {error, Reason}
2885     end;
2886 parse_phpmod(['<', "extern", ',', NodeModFunSpec, '>'], _) ->
2887     case string_to_node_mod_fun(NodeModFunSpec) of
2888     {ok, Node, Mod, Fun} ->
2889         {ok, {extern, {Node, Mod, Fun}}};
2890     {ok, Mod, Fun} ->
2891         {ok, {extern, {Mod, Fun}}};
2892     {error, Reason} ->
2893         {error, Reason}
2894     end.
2895
2896
2897 parse_compressible_mime_types(_, all) ->
2898 {ok, all};
2899 parse_compressible_mime_types(["all" | _], _Acc) ->
2900 {ok, all};
2901 parse_compressible_mime_types(["defaults" | Rest], Acc) ->
2902     parse_compressible_mime_types(Rest, ?DEFAULT_COMPRESSIBLE_MIME_TYPES++Acc);
2903 parse_compressible_mime_types(['.' | Rest], Acc) ->
2904     parse_compressible_mime_types(Rest, Acc);
2905 parse_compressible_mime_types([MimeType | Rest], Acc) ->
2906     case parse_mime_type(MimeType) of
2907     {ok, "*", "*"} ->
2908         {ok, all};
2909     {ok, Type, "*"} ->
2910         parse_compressible_mime_types(Rest, [{Type, all} | Acc]);
2911     {ok, Type, SubType} ->
2912         parse_compressible_mime_types(Rest, [{Type, SubType} | Acc]);
2913     Error ->
2914         Error
2915     end;
2916 parse_compressible_mime_types([], Acc) ->
2917 {ok, Acc}.
2918
2919
2920 parse_mime_type(MimeType) ->

```

```

2921     Res = re:run(MimeType, "^([-\\w\\+|\\*|\\/|\\.|\\+|\\*|\\$)",
2922                 [{capture, all_but_first, list}]),
2923     case Res of
2924     {match, [Type, SubType]} ->
2925     {ok, Type, SubType};
2926     nomatch ->
2927     {error, "Invalid MimeType"}
2928     end.
2929
2930
2931 parse_index_files([]) ->
2932 ok;
2933 parse_index_files([Idx|Rest]) ->
2934 case Idx of
2935     [_|_] when Rest /= [] ->
2936     {error, "Only the last index should be absolute"};
2937     _ ->
2938     parse_index_files(Rest)
2939 end.
2940
2941 is_valid_mime_type(MimeType) ->
2942 case re:run(MimeType, "^([-\\w\\+|\\/|\\.|\\+|\\*|\\$)", [{capture, none}]] of
2943     match -> true;
2944     nomatch -> false
2945 end.
2946
2947 parse_mime_types(['<', MimeType, '| Tail], Acc0) ->
2948     Exts = lists:takewhile(fun(X) -> X /= '>' end, Tail),
2949     [_|Tail2] = lists:dropwhile(fun(X) -> X /= '>' end, Tail),
2950     Acc1 = lists:foldl(fun(E, Acc) ->
2951         lists:keystore(E, 1, Acc, {E, MimeType})
2952     end, Acc0, Exts),
2953     case is_valid_mime_type(MimeType) of
2954     true -> parse_mime_types(Tail2, Acc1);
2955     false -> {error, ?F("Invalid mime-type '~p'", [MimeType])}
2956     end;
2957 parse_mime_types([], Acc)->
2958 {ok, lists:reverse(Acc)};
2959 parse_mime_types(_, _) ->
2960 {error, "Unexpected tokens"}.
2961
2962 parse_charsets(['<', Charset, '| Tail], Acc0) ->
2963     Exts = lists:takewhile(fun(X) -> X /= '>' end, Tail),
2964     [_|Tail2] = lists:dropwhile(fun(X) -> X /= '>' end, Tail),
2965     Acc1 = lists:foldl(fun(E, Acc) ->
2966         lists:keystore(E, 1, Acc, {E, Charset})
2967     end, Acc0, Exts),
2968     parse_charsets(Tail2, Acc1);
2969 parse_charsets([], Acc)->
2970 {ok, lists:reverse(Acc)};
2971 parse_charsets(_, _) ->
2972 {error, "Unexpected tokens"}.
2973
2974
2975 parse_mime_types_info(Directive, Type, undefined, undefined) ->
2976     parse_mime_types_info(Directive, Type, #mime_types_info{});
2977 parse_mime_types_info(Directive, Type, undefined, DefaultInfo) ->
2978     parse_mime_types_info(Directive, Type, DefaultInfo);
2979 parse_mime_types_info(Directive, Type, Info, _) ->
2980     parse_mime_types_info(Directive, Type, Info).
2981
2982 parse_mime_types_info(default_type, Type, Info) ->
2983     case is_valid_mime_type(Type) of
2984     true -> {ok, Info#mime_types_info{default_type=Type}};
2985     false -> {error, ?F("Invalid mime-type '~p'", [Type])}
2986     end;
2987 parse_mime_types_info(default_charset, Charset, Info) ->
2988     {ok, Info#mime_types_info{default_charset=Charset}};
2989 parse_mime_types_info(mime_types_file, File, Info) ->
2990     {ok, Info#mime_types_info{mime_types_file=File}};
2991 parse_mime_types_info(add_types, NewTypes, Info) ->
2992     case parse_mime_types(NewTypes, Info#mime_types_info.types) of
2993     {ok, Types} -> {ok, Info#mime_types_info{types=Types}};
2994     Error -> Error
2995     end;
2996 parse_mime_types_info(add_charsets, NewCharsets, Info) ->
2997     case parse_charsets(NewCharsets, Info#mime_types_info.charsets) of
2998     {ok, Charsets} -> {ok, Info#mime_types_info{charsets=Charsets}};
2999     Error -> Error
3000     end.
3001
3002
3003 parse_nslookup_pref(Pref) ->
3004     parse_nslookup_pref(Pref, []).
3005
3006 parse_nslookup_pref(Empty, []) when Empty == [] orelse Empty == ['.', '.'] ->
3007     %% Get default value, if nslookup_pref = [].
3008     {ok, yaws:gconf_nslookup_pref(#gconf{}));
3009 parse_nslookup_pref([C, Family | Rest], Result)
3010 when C == '[' orelse C == ',' ->
3011     case Family of
3012     "inet" ->
3013         case lists:member(inet, Result) of
3014             false -> parse_nslookup_pref(Rest, [inet | Result]);
3015             true -> parse_nslookup_pref(Rest, Result)
3016         end;
3017     "inet6" ->
3018         case lists:member(inet6, Result) of

```

```

3019         false -> parse_nslookup_pref(Rest, [inet6 | Result]);
3020         true  -> parse_nslookup_pref(Rest, Result)
3021     end;
3022 - ->
3023     case Result of
3024     [PreviousFamily | _] ->
3025         {error, ?F("Invalid nslookup_pref: invalid family or "
3026             "token '~s', after family '~s'",
3027             [Family, PreviousFamily])};
3028     [] ->
3029         {error, ?F("Invalid nslookup_pref: invalid family or "
3030             "token '~s'", [Family])}
3031     end
3032 end;
3033 parse_nslookup_pref([''], Result) ->
3034 {ok, lists:reverse(Result)};
3035 parse_nslookup_pref([Invalid | _], []) ->
3036 {error, ?F("Invalid nslookup_pref: unexpected token '~s'", [Invalid])};
3037 parse_nslookup_pref([Invalid | _], [Family | _]) ->
3038 {error, ?F("Invalid nslookup_pref: unexpected token '~s', "
3039     "after family '~s'", [Invalid, Family])}.
3040
3041
3042 parse_redirect(Path, [Code, URL], Mode, Lno) ->
3043     case catch list_to_integer(Code) of
3044     I when is_integer(I), I >= 300, I <= 399 ->
3045         try yaws_api:parse_url(URL, sloppy) of
3046         U when is_record(U, url) ->
3047             {Path, I, U, Mode}
3048         catch _:_ ->
3049             {error, ?F("Bad redirect URL ~p at line ~w", [URL, Lno])}
3050         end;
3051     I when is_integer(I), I >= 100, I <= 599 ->
3052         %% Only relative path are authorized here
3053         try yaws_api:parse_url(URL, sloppy) of
3054         #url{scheme=undefined, host=[], port=undefined, path=P} ->
3055             {Path, I, P, Mode};
3056         #url{} ->
3057             {error, ?F("Bad redirect rule at line ~w: "
3058                 "Absolute URL is forbidden here", [Lno])}
3059         catch _:_ ->
3060             {error, ?F("Bad redirect URL ~p at line ~w", [URL, Lno])}
3061         end;
3062     _ ->
3063         {error, ?F("Bad status code ~p at line ~w", [Code, Lno])}
3064     end;
3065 parse_redirect(Path, [CodeOrUrl], Mode, Lno) ->
3066     case catch list_to_integer(CodeOrUrl) of
3067     I when is_integer(I), I >= 300, I <= 399 ->
3068         {error, ?F("Bad redirect rule at line ~w: "
3069             "URL to redirect to is missing ", [Lno])};
3070     I when is_integer(I), I >= 100, I <= 599 ->
3071         {Path, I, undefined, Mode};
3072     I when is_integer(I) ->
3073         {error, ?F("Bad status code ~p at line ~w", [CodeOrUrl, Lno])};
3074     _ ->
3075         try yaws_api:parse_url(CodeOrUrl, sloppy) of
3076         #url{}=U ->
3077             {Path, 302, U, Mode}
3078         catch _:_ ->
3079             {error, ?F("Bad redirect URL ~p at line ~w",
3080                 [CodeOrUrl, Lno])}
3081         end
3082     end;
3083 parse_redirect(_Path, _, _Mode, Lno) ->
3084 {error, ?F("Bad redirect rule at line ~w", [Lno])}.
3085
3086
3087 ssl_start() ->
3088     case catch ssl:start() of
3089     ok ->
3090         ok;
3091     {error, {already_started, ssl}} ->
3092         ok;
3093     Err ->
3094         error_logger:format("Failed to start ssl: ~p~n", [Err])
3095     end.
3096
3097
3098
3099 %% search for an SC within Pairs that have the same, listen,port,ssl,servername
3100 %% Return {Pid, SC, Scs} or false
3101 %% Pairs is the pairs in yaws_server #state{}
3102 search_sconf(GC, NewSC, Pairs) ->
3103     case lists:zf(
3104         fun({Pid, Scs = [SC|_]}) ->
3105             case same_virt_srv(GC, NewSC, SC) of
3106             true ->
3107                 case lists:keysearch(NewSC#sconf.servername,
3108                     #sconf.servername, Scs) of
3109                     {value, Found} ->
3110                         {true, {Pid, Found, Scs}};
3111                     false ->
3112                         false
3113                     end;
3114                 false ->
3115                     false
3116             end
3117         end

```



```

3118         en, Pairs) of
3119         [] ->
3120             false;
3121         [{Pid, Found, Scs}] ->
3122             {Pid, Found, Scs};
3123         _Other ->
3124             error_logger:format("Fatal error, no two sconfs should "
3125                                 "ever be considered equal ..",[]),
3126             erlang:error(fatal_conf)
3127     end.
3128
3129 %% find the group a new SC would belong to
3130 search_group(Gc, SC, Pairs) ->
3131     Fun = fun({Pid, [S|Ss]}) ->
3132         case same_virt_srv(Gc, S, SC) of
3133             true ->
3134                 {true, {Pid, [S|Ss]}};
3135             false ->
3136                 false
3137         end
3138     end,
3139     lists:zf(Fun, Pairs).
3140
3141
3142 %% Return a new Pairs list with one SC updated
3143 update_sconf(Gc, NewSc, Pos, Pairs) ->
3144     lists:map(
3145         fun({Pid, Scs}) ->
3146             case same_virt_srv(Gc, hd(Scs), NewSc) of
3147                 true ->
3148                     L2 = lists:keydelete(NewSc#sconf.servername,
3149                                           #sconf.servername, Scs),
3150                     (Pid, yaws:insert_at(NewSc, Pos, L2));
3151                 false ->
3152                     {Pid, Scs}
3153             end
3154         end, Pairs).
3155
3156
3157 %% return a new pairs list with SC removed
3158 delete_sconf(Gc, OldSc, Pairs) ->
3159     lists:zf(
3160         fun({Pid, Scs}) ->
3161             case same_virt_srv(Gc, hd(Scs), OldSc) of
3162                 true ->
3163                     L2 = lists:keydelete(OldSc#sconf.servername,
3164                                           #sconf.servername, Scs),
3165                     {true, {Pid, L2}};
3166                 false ->
3167                     {true, {Pid, Scs}}
3168             end
3169         end, Pairs).
3170
3171
3172
3173
3174 same_virt_srv(Gc, S, NewSc) when S#sconf.listen == NewSc#sconf.listen,
3175                               S#sconf.port == NewSc#sconf.port ->
3176     if
3177         Gc#gconf.sni == disable orelse
3178         S#sconf.ssl == undefined orelse
3179         NewSc#sconf.ssl == undefined ->
3180             (S#sconf.ssl == NewSc#sconf.ssl);
3181         true ->
3182             true
3183     end;
3184     same_virt_srv(_,_,_) ->
3185         false.
3186
3187
3188 eq_sconfs(S1,S2) ->
3189     (S1#sconf.port == S2#sconf.port andalso
3190      S1#sconf.flags == S2#sconf.flags andalso
3191      S1#sconf.redirect_map == S2#sconf.redirect_map andalso
3192      S1#sconf.rhost == S2#sconf.rhost andalso
3193      S1#sconf.rmethod == S2#sconf.rmethod andalso
3194      S1#sconf.docroot == S2#sconf.docroot andalso
3195      S1#sconf.xtra_docroots == S2#sconf.xtra_docroots andalso
3196      S1#sconf.listen == S2#sconf.listen andalso
3197      S1#sconf.servername == S2#sconf.servername andalso
3198      S1#sconf.yaws == S2#sconf.yaws andalso
3199      S1#sconf.ssl == S2#sconf.ssl andalso
3200      S1#sconf.authdirs == S2#sconf.authdirs andalso
3201      S1#sconf.partial_post_size == S2#sconf.partial_post_size andalso
3202      S1#sconf.appmods == S2#sconf.appmods andalso
3203      S1#sconf.expires == S2#sconf.expires andalso
3204      S1#sconf.errormod_401 == S2#sconf.errormod_401 andalso
3205      S1#sconf.errormod_404 == S2#sconf.errormod_404 andalso
3206      S1#sconf.errormod_crash == S2#sconf.errormod_crash andalso
3207      S1#sconf.arg_rewrite_mod == S2#sconf.arg_rewrite_mod andalso
3208      S1#sconf.logger_mod == S2#sconf.logger_mod andalso
3209      S1#sconf.opaque == S2#sconf.opaque andalso
3210      S1#sconf.start_mod == S2#sconf.start_mod andalso
3211      S1#sconf.allowed_scripts == S2#sconf.allowed_scripts andalso
3212      S1#sconf.tilde_allowed_scripts == S2#sconf.tilde_allowed_scripts andalso
3213      S1#sconf.index_files == S2#sconf.index_files andalso
3214      S1#sconf.revproxy == S2#sconf.revproxy andalso

```

```

3215     S1#sconf.options == S2#sconf.options andalso
3216     S1#sconf.extra CGI vars == S2#sconf.extra CGI vars andalso
3217     S1#sconf.stats == S2#sconf.stats andalso
3218     S1#sconf.fcgi_app_server == S2#sconf.fcgi_app_server andalso
3219     S1#sconf.php_handler == S2#sconf.php_handler andalso
3220     S1#sconf.shaper == S2#sconf.shaper andalso
3221     S1#sconf.deflate_options == S2#sconf.deflate_options andalso
3222     S1#sconf.mime_types_info == S2#sconf.mime_types_info andalso
3223     S1#sconf.dispatch_mod == S2#sconf.dispatch_mod andalso
3224     S1#sconf.extra_response_headers == S2#sconf.extra_response_headers).
3225
3226 %% This is the version of setconf that performs a
3227 %% soft reconfig, it requires the args to be checked.
3228 soft_setconf(GC, Groups, OLDGC, OldGroups) ->
3229     if
3230         GC /= OLDGC ->
3231             yaws_trace:setup(GC),
3232             update_gconf(GC);
3233         true ->
3234             ok
3235     end,
3236     compile_and_load_src_dir(GC),
3237     Grps = load_mime_types_module(GC, Groups),
3238     Rems = remove_old_scs(GC, lists:flatten(OldGroups), Grps),
3239     Adds = soft_setconf_scs(GC, lists:flatten(Grps), 1, OldGroups),
3240     lists:foreach(
3241         fun({delete_sconf, SC}) ->
3242             delete_sconf(SC);
3243         ({add_sconf, N, SC}) ->
3244             add_sconf(N, SC);
3245         ({update_sconf, N, SC}) ->
3246             update_sconf(N, SC)
3247     end, Rems ++ Adds).
3248
3249
3250
3251 hard_setconf(GC, Groups) ->
3252     gen_server:call(yaws_server,{setconf, GC, Groups}, infinity).
3253
3254
3255 remove_old_scs(Gc, [Sc|Scs], NewGroups) ->
3256     case find_group(Gc, Sc, NewGroups) of
3257         false ->
3258             [{delete_sconf, Sc} | remove_old_scs(Gc, Scs, NewGroups)];
3259         {true, G} ->
3260             case find_sc(Sc, G) of
3261                 false ->
3262                     [{delete_sconf, Sc} | remove_old_scs(Gc, Scs, NewGroups)];
3263                 _ ->
3264                     remove_old_scs(Gc, Scs, NewGroups)
3265             end
3266     end;
3267 remove_old_scs(_, [], _) ->
3268     [].
3269
3270 soft_setconf_scs(Gc, [Sc|Scs], N, OldGroups) ->
3271     case find_group(Gc, Sc, OldGroups) of
3272         false ->
3273             [{add_sconf,N,Sc} | soft_setconf_scs(Gc, Scs, N+1, OldGroups)];
3274         {true, G} ->
3275             case find_sc(Sc, G) of
3276                 false ->
3277                     [{add_sconf,N,Sc} | soft_setconf_scs(Gc, Scs,N+1,OldGroups)];
3278                 {true, _OldSc} ->
3279                     [{update_sconf,N,Sc} | soft_setconf_scs(Gc, Scs,N+1,OldGroups)]
3280             end
3281     end;
3282 soft_setconf_scs(_, [], _, _) ->
3283     [].
3284
3285
3286 %% checking code
3287
3288 can_hard_gc(New, Old) ->
3289     if
3290         Old == undefined ->
3291             true;
3292         New#gconf.yaws_dir == Old#gconf.yaws_dir,
3293         New#gconf.runmods == Old#gconf.runmods,
3294         New#gconf.logdir == Old#gconf.logdir ->
3295             true;
3296         true ->
3297             false
3298     end.
3299
3300
3301
3302 can_soft_setconf(NEWGC, NewGroups, OLDGC, OldGroups) ->
3303     can_soft_gc(NEWGC, OLDGC) andalso
3304     can_soft_sconf(NEWGC, lists:flatten(NewGroups), OldGroups).
3305
3306 can_soft_gc(G1, G2) ->
3307     if
3308         G1#gconf.flags == G2#gconf.flags,
3309         G1#gconf.logdir == G2#gconf.logdir,
3310         G1#gconf.log_wrap_size == G2#gconf.log_wrap_size,
3311         G1#gconf.sni == G2#gconf.sni,
3312         G1#gconf.id == G2#gconf.id ->

```

```

3313         true;
3314     true ->
3315         false
3316     end.
3317
3318
3319 can_soft_sconf(Gc, [Sc|Scs], OldGroups) ->
3320     case find_group(Gc, Sc, OldGroups) of
3321         false ->
3322             can_soft_sconf(Gc, Scs, OldGroups);
3323         {true, G} ->
3324             case find_sc(Sc, G) of
3325                 false ->
3326                     can_soft_sconf(Gc, Scs, OldGroups);
3327                 {true, Old} when Old#sconf.start_mod /= Sc#sconf.start_mod ->
3328                     false;
3329                 {true, Old} ->
3330                     case
3331                         {proplists:get_value(listen_opts, Old#sconf.soptions),
3332                          proplists:get_value(listen_opts, Sc#sconf.soptions)} of
3333                         {Opts, Opts} ->
3334                             can_soft_sconf(Gc, Scs, OldGroups);
3335                         _ ->
3336                             false
3337                     end
3338                 end
3339             end;
3340     can_soft_sconf(_, [], _) ->
3341         true.
3342
3343
3344 find_group(GC, SC, [G|Gs]) ->
3345     case same_virt_srv(GC, SC, hd(G)) of
3346         true ->
3347             {true, G};
3348         false ->
3349             find_group(GC, SC, Gs)
3350     end;
3351 find_group(_,_,[]) ->
3352     false.
3353
3354 find_sc(SC, [S|Ss]) ->
3355     if SC#sconf.servername == S#sconf.servername ->
3356         {true, S};
3357     true ->
3358         find_sc(SC, Ss)
3359     end;
3360 find_sc(SC,[]) ->
3361     false.
3362
3363
3364 verify_upgrade_args(GC, Groups0) when is_record(GC, gconf) ->
3365     SCs0 = lists:flatten(Groups0),
3366     case lists:all(fun(SC) -> is_record(SC, sconf) end, SCs0) of
3367         true ->
3368             %% Embedded code may give appmods as a list of strings, or
3369             %% appmods can be {StringPathElem,ModAtom} or
3370             %% {StringPathElem,ModAtom,ExcludePathsList} tuples. Handle
3371             %% all possible variants here.
3372             SCs1 = lists:map(
3373                 fun(SC) ->
3374                     SC#sconf{appmods =
3375                         lists:map(
3376                             fun({PE, Mod}) ->
3377                                 {PE, Mod};
3378                             ({PE,Mod,Ex}) ->
3379                                 {PE,Mod,Ex};
3380                             (AM) when is_list(AM) ->
3381                                 {AM,list_to_atom(AM)};
3382                             (AM) when is_atom(AM) ->
3383                                 {atom_to_list(AM), AM}
3384                         end,
3385                         SC#sconf.appmods)}
3386             end, SCs0),
3387         case catch validate_cs(GC, SCs1) of
3388             {ok, GC, Groups1} -> {GC, Groups1};
3389             {error, Reason} -> erlang:error(Reason);
3390             _ -> erlang:error(badgroups)
3391         end;
3392         false ->
3393             erlang:error(badgroups)
3394     end.
3395
3396
3397
3398 add_sconf(SC) ->
3399     add_sconf(-1, SC).
3400
3401
3402 add_sconf(Pos, SC0) ->
3403     {ok, SC1} = gen_server:call(yaws_server, {add_sconf, Pos, SC0}, infinity),
3404     ok = yaws_log:add_sconf(SC1),
3405     {ok, SC1}.
3406
3407
3408 update_sconf(Pos, SC) ->
3409     gen_server:call(yaws_server, {update_sconf, Pos, SC}, infinity).
3410
3411
3412 delete_sconf(SC) ->
3413     ok = gen_server:call(yaws_server, {delete_sconf, SC}, infinity),

```

```

3411     ok = yaws_log:del_sconf(SC).
3412
3413 update_gconf(GC) ->
3414     ok = gen_server:call(yaws_server, {update_gconf, GC}, infinity).
3415
3416
3417 parse_auth_ips([], Result) ->
3418     Result;
3419 parse_auth_ips([Str|Rest], Result) ->
3420     try
3421         parse_auth_ips(Rest, [yaws:parse_ipmask(Str)|Result])
3422     catch
3423         _:_ -> parse_auth_ips(Rest, Result)
3424     end.
3425
3426 parse_auth_user(User, Lno) ->
3427     try
3428         [Name, Passwd] = string:tokens(User, ":"),
3429         case re:run(Passwd, "([^\s]+)?(?:\\$([^\s]+\s)?(?:.+), [{capture,all_but_first,list}]) of
3430             {match, [Algo, B64Salt, B64Hash]} ->
3431                 case parse_auth_user(Name, Algo, B64Salt, B64Hash) of
3432                     {ok, Res} ->
3433                         Res;
3434                     {error, bad_algo} ->
3435                         {error, ?F("Unsupported hash algorithm '~p' at line ~w",
3436                             [Algo, Lno])};
3437                     {error, bad_user} ->
3438                         {error, ?F("Invalid user at line ~w", [Lno])}
3439                 end;
3440         _:_ ->
3441             Salt = crypto:strong_rand_bytes(32),
3442             {Name, sha256, Salt, crypto:hash(sha256, [Salt, Passwd])}
3443     end
3444 catch
3445     _:_ ->
3446         {error, ?F("Invalid user at line ~w", [Lno])}
3447 end.
3448
3449 parse_auth_user(User, Algo, B64Salt, B64Hash) ->
3450     try
3451         if
3452             Algo == "md5" orelse Algo == "sha" orelse
3453             Algo == "sha224" orelse Algo == "sha256" orelse
3454             Algo == "sha384" orelse Algo == "sha512" orelse
3455             Algo == "ripemd160" ->
3456             Salt = base64:decode(B64Salt),
3457             Hash = base64:decode(B64Hash),
3458             {ok, {User, list_to_atom(Algo), Salt, Hash}};
3459             true ->
3460             {error, bad_algo}
3461         end
3462     catch
3463         _:_ -> {error, bad_user}
3464     end.
3465
3466
3467 subconfigfiles(FD, Name, Lno) ->
3468     {ok, Config} = file:pid2name(FD),
3469     ConfPath = filename:dirname(filename:absname(Config)),
3470     File = filename:absname(Name, ConfPath),
3471     case {is_file(File), is_wildcard(Name)} of
3472         {true,_} ->
3473             {ok, [File]};
3474         {false,true} ->
3475             Names = filelib:wildcard(Name, ConfPath),
3476             Files = [filename:absname(N, ConfPath) || N <- lists:sort(Names)],
3477             {ok, lists:filter(fun filter_subconfigfile/1, Files)};
3478         {false,false} ->
3479             {error, ?F("Expect filename or wildcard at line ~w"
3480                 " (subconfig: ~s)", [Lno, Name])}
3481     end.
3482
3483 subconfigdir(FD, Name, Lno) ->
3484     {ok, Config} = file:pid2name(FD),
3485     ConfPath = filename:dirname(filename:absname(Config)),
3486     Dir = filename:absname(Name, ConfPath),
3487     case is_dir(Dir) of
3488         true ->
3489             case file:list_dir(Dir) of
3490                 {ok, Names} ->
3491                     Files = [filename:absname(N, Dir) || N <- lists:sort(Names)],
3492                     {ok, lists:filter(fun filter_subconfigfile/1, Files)};
3493                 {error, Error} ->
3494                     {error, ?F("Directory ~s is not readable: ~s",
3495                         [Name, Error])}
3496             end;
3497         false ->
3498             {error, ?F("Expect directory at line ~w (subconfigdir: ~s)",
3499                 [Lno, Dir])}
3500     end.
3501
3502 filter_subconfigfile(File) ->
3503     case filename:basename(File) of
3504         [$_|_] ->
3505             error_logger:info_msg("Yaws: Ignore subconfig file ~s~n", [File]),
3506             false;
3507         _ ->
3508             true
3509     end

```

```

3509     end.
3510
3511 fload_subconfigfiles([], global, GC, Cs) ->
3512     {ok, GC, Cs};
3513 fload_subconfigfiles([File|Files], global, GC, Cs) ->
3514     error_logger:info_msg("Yaws: Using global subconfig file ~s~n", [File]),
3515     case file:open(File, [read]) of
3516     {ok, FD} ->
3517         R = (catch fload(FD, GC, Cs, 1, ?NEXTLINE)),
3518         ?Debug("FLOAD(~s): ~p", [File, R]),
3519         case R of
3520         {ok, GC1, Cs1} -> fload_subconfigfiles(Files, global, GC1, Cs1);
3521         Err -> Err
3522         end;
3523     Err ->
3524         {error, ?F("Can't open subconfig file ~s: ~p", [File,Err])}
3525     end;
3526 fload_subconfigfiles([], server, GC, C) ->
3527     {ok, GC, C};
3528 fload_subconfigfiles([File|Files], server, GC, C) ->
3529     error_logger:info_msg("Yaws: Using server subconfig file ~s~n", [File]),
3530     case file:open(File, [read]) of
3531     {ok, FD} ->
3532         R = (catch fload(FD, server, GC, C, 1, ?NEXTLINE)),
3533         ?Debug("FLOAD(~s): ~p", [File, R]),
3534         case R of
3535         {ok, GC1, C1, _, eof} ->
3536             fload_subconfigfiles(Files, server, GC1, C1);
3537         {ok, _, _, Lno, ['<', "/server", '>']} ->
3538             {error, ?F("Unexpected closing tag in subconfgile ~s"
3539                 " at line ~w ", [File, Lno])};
3540         Err ->
3541             Err
3542         end;
3543     Err ->
3544         {error, ?F("Can't open subconfig file ~s: ~p", [File,Err])}
3545     end.
3546
3547
3548 str2term(Str0) ->
3549     Str=Str0++".",
3550     {ok,Tokens,_EndLine} = erl_scan:string(Str),
3551     {ok,AbsForm} = erl_parse:parse_exprs(Tokens),
3552     {value,Value,_Bs} = erl_eval:exprs(AbsForm, erl_eval:new_bindings()),
3553     Value.
3554
3555 check_ciphers([], _) ->
3556     ok;
3557 check_ciphers([Spec|Specs], L) ->
3558     case lists:member(Spec, L) of
3559     true ->
3560         check_ciphers(Specs, L);
3561     false ->
3562         {error, ?F("Bad cipherspec ~p",[Spec])}
3563     end;
3564 check_ciphers(X,_) ->
3565     {error, ?F("Bad cipherspec ~p",[X])}.
3566
3567 check_eccs(From_conf, Available) ->
3568     case From_conf -- Available of
3569     [] -> ok;
3570     Bad -> {error, ?F("Bad elliptic curves ~p",[Bad])}
3571     end.
3572
3573 io_get_line(FD, Prompt, Acc) ->
3574     Next = io:get_line(FD, Prompt),
3575     if
3576     is_list(Next) ->
3577         case lists:reverse(Next) of
3578         [$_n, $_\ | More] ->
3579             io_get_line(FD, Prompt, Acc ++ lists:reverse(More));
3580         _ ->
3581             Acc ++ Next
3582         end;
3583     true ->
3584         Next
3585     end.
3586
3587 update_options(SC, Name, Key, Value) ->
3588     Opts0 = proplists:get_value(Name, SC#sconf.soptions),
3589     Opts1 = lists:keystore(Key, 1, Opts0, {Key, Value}),
3590     SOpts = lists:keystore(Name, 1, SC#sconf.soptions, {Name, Opts1}),
3591     SC#sconf{soptions = SOpts}.
3592
3593 set_sendfile_flags(GC, "erlang") ->
3594     {ok, ?gc_set_use_erlang_sendfile(GC, true)};
3595 set_sendfile_flags(GC, "disable") ->
3596     {ok, ?gc_set_use_erlang_sendfile(GC, false)};
3597 set_sendfile_flags(_, _) ->
3598     {error, "Expect erlang|disable"}.

```