

## 1 Introduction

MinaliC is a small web server that is developed in the C programming language.

Currently it has been developed for the Windows platform.

The server binary and source code can be downloaded from:

**<http://sourceforge.net/projects/minalic/>.**

MinaliC is intended to be a minimal server to be used for small web applications, when full scale web server functionalities is not needed. The server has support for calling CGI scripts (Common Gateway Interface). It also has support for calling PHP and Perl scripts. MinaliC can be run either as a windows service or as a console application.

The server only has one executable binary - minalic.exe.

MinaliC can be expanded by building plug-ins which implements certain protocols, so the server can be more than just a web server. It can be a server for your own protocol. Multiple instances of the server can be installed, each listening to a certain port. And each instance can listen on several ports.

## 2 Directory structure

The MinaliC binary can be placed in any directory on the disk. In this document that directory will be called <Binary directory>. In this directory there must be a subdirectory called wwwroot. That is the root directory of all web applications.

In the wwwroot you can create subdirectories that will act as web-directories.

Example of directory structure:

```
<Binary directory>
|
minalic.exe
minalic.ini
minalic.log
<wwwroot>
|
index.htm
logo.bmp
<subdir>
|
index.cgi
<subdir2>
|
index.php
```

The web default file - the file chosen if no filename is specified - is index.cgi, index.php index.pl and index.htm (in priority order from the highest).

### 3 Configuration

The configuration file is called minalic.ini and is placed in the <Binary directory>.

Configuration variables are:

- HTTP\_PORT, sets the port that the server listens to. Default value is 80.
- SERVER\_LOGLEVEL is the level of how much of the server activities that is logged. Default value is 1. With a value of 2 the server will log http request and response headers.
- SESSION\_TIMEOUT is the number of seconds that a session is valid. Default value is 1800 (30 minutes).
- SERVER\_HANDLERS is the number of simultaneous client handler threads that are allowed. Default value is 100.
- CGI\_TIMEOUT is the number of seconds that a CGI module can be running without returning any data. Default value is 90 seconds.
- SERVER\_PLUGIN loads a plug-in module and binds it to a port number.
- DIRECTORY\_BROWSING\_DEFAULT, decides if directory browsing is allowed for all directories (1) or only directories with a dir.ini file (0). Default value is 0.

For example : HTTP\_PORT = 8080.

Comment out a line with a beginning asterisk (\*).

### 4 Logging

The name of the log file is minalic.log and it will be created in the <Binary directory>.

When the server is started as a console application, all logging will be done through the standard output.

### 5 Running

To install the server as a windows service type minalic.exe -install. The created service will be started and set to start automatically. To uninstall the service type minalic.exe -uninstall. With the flags -start and -stop the service can be set to start or stop.

To specify the name of the service when installed type minalic.exe -install -name:my\_name.

The -name: option can be used on all of the commands -install, -uninstall, -start and -stop.

Example: minalic.exe -stop -name:my\_service.

Starting the binary minalic.exe with no parameters will start the server as a console application.

## 6 Directory browsing

By default directory browsing is turned off. It can be turned on by setting the configuration value `DIRECTORY_BROWSING_DEFAULT` to 1. If this value is set to 0, directory browsing is by default not allowed in any directory. But directory browsing can be turned on in a specific directory by putting an empty file called `dir.ini` in the specific directory. No directory will be browsed that contain an index file. Then instead the index file will be returned.

## 7 HTTP specific

### 7.1 Requests

MinaliC supports the request methods GET and POST. For other requests the server will respond with a 501 Not implemented.

### 7.2 Response

A response from the server can be any of these:

- 200 OK
- 501 Not implemented
- 404 Not Found
- 400 Bad Request
- 301 Moved Permanently
- 302 Redirect
- 500 Internal Server Error
- 403 Not Modified

HTTP header fields that MinaliC sends in a response are:

- The HTTP status line (example HTTP/1.1 200 OK)
- Date:
- Content-Length:
- Content-Type:
- Last-Modified:
- Server:
- Connection:
- Set-Cookie:

The server can recognize 46 file extensions and decide the correct mime type to return in the Content-Type header.

If a request includes the header `If-Modified-Since` it looks to see if the requested file has been updated since last call.

## 8 Sessions

MinaliC handles sessions by help of a sessionid that is included in a domain cookie. For each request the server gets it will respond with a valid sessionid. The timeout of a session occurs when there has not been any request for a specific amount of time. That time is set in the configuration file with the variable `SERVER_TIMEOUT`. If the server finds out that the sessionid returned by the browser

is timed out it will create a new sessionid. A sessionid is a 30 byte alfa numeric string.

## 9 CGI

A CGI module is a file with the extension .cgi. The server will try to load the corresponding file with the extension .exe. CGI modules can be put in the same directory as the HTML pages. For example the request <http://localhost/subdirectory/myscript.cgi> will invoke the executable myscript.exe that is put in the wwwroot\subdirectory directory.

When a CGI module is invoked the server sends the input content to the module through standard input and it receives the output from the CGI module back from the modules standard output.

The CGI module can send the following headers back to the server:

- Location:
- Content-Type:
- Set-Cookie:

If the script sends an absolute location address in a Location header (starting with http://) the server will send a 302 Redirect to the client. If the script sends a local address the server will handle the redirect by itself by responding with the new local page that was requested.

The CGI module will receive the following headers from the server through the environment variables:

```
REQUEST_METHOD
QUERY_STRING
SERVER_SOFTWARE
SERVER_NAME
SCRIPT_NAME
SCRIPT_FILENAME
GATEWAY_INTERFACE
SERVER_PROTOCOL
REMOTE_ADDR
SESSION_ID
WWWROOT (physical path to the wwwroot directory for the server)
HTTP_ACCEPT
CONTENT_TYPE
CONTENT_LENGTH
HTTP_COOKIE
```

## 10 PHP and Perl

MinaliC also has support for running PHP and Perl scripts. To run this kind of scripts from the server, PHP or Perl first must be installed on the server. MinaliC calls PHP and Perl as a CGI module and communicates with the interpreter as with ordinary CGI modules. The PHP executable must be named php.exe, and the Perl executable must be named perl.exe.

## 11 Internal structure and source code

MinaliC is a multi threaded server. Each request will end up in a handler thread. For safety reasons the number of simultaneous threads can be limited by the configuration parameter `SERVER_HANDLERS`. If the number of requests exceeds the limit the server will rejected the requests.

The source code is developed in an object oriented manner - but in C. Each class has a header file (.h) and an implementation file (.c).

## 12 Plug-ins

MinaliC has support for loading custom plug-ins to handle all communication on a specific port. In the configuration file it is possible to declare several plug-ins on the form:

`PLUGIN = {plugin_name}:{port}`. A plug-in is a dll file that implements some predefined functions. These functions are called from MinaliC at certain points. For example you could build a plug-in that implements some protocol that is based on TCP/IP. The {plugin\_name} is the name of the plug-in dll with the .dll excluded. The {port} is the TCP/IP port to listen on.

It is possible to declare several plugging.

For example:

`PLUGIN = FTP:21`

`PLUGIN = LDAP:78.`

In this example the server looks for the two dlls' ftp.dll and ldap.dll. These dlls' should be put in the <Binary directory>.

The implementation of a plug-in dll is based on exporting three functions. These functions are:

1. `char *get_version()`
2. `char *get_build()`
3. `void handle_request(SOCKET s, struct sockaddr_in addr)`

Functions 1 and 2 are very simple. The only purpose of these functions is to return the version and the build id of the plug-in. These values are stings and can be decided by the implementer. Function number 3 is the actual implementation of the protocol handler for the protocol that the plug-in is supposed to implement. MinaliC sends two parameters to the function. These parameters are the socket to read from and send to for the connection and also an address parameter including the clients address.

The function is called from within a handler thread in MinaliC, and each connection on the specific port will end up in a call to this function. This makes the plug-in dll multi threaded. By implementing this function the plug-in therefore can handle several multiple connections at the same time.

Here is a very simple implementation of a plug-in dll in Visual Studio:

```
__declspec(dllexport) char *get_version()
{
    return "2.1";
}

__declspec(dllexport) char *get_build()
{
    return "PL0023";
}

/*This protocol is very simple. It waits for 4 characters from the client and
then sends the string "My return string" back to the client.*/
__declspec(dllexport) void handle_request(SOCKET s,struct sockaddr_in addr)
{
    int len;
    char header[256];
    int max;
    char *answ="My return string";

    max = 0;
    while (max < 4)
    {
        len = recv(s,&header[max],1,0);
        if (len == 0)
            return ; /* No data found*/

        if (len < 0)
            return; /* Socket closed*/
        max += len;
    }
    header[max+1] = 0;
    send(s,answ,strlen(answ),0);
}
```

It is by help of plug-ins possible to make MinaliC handle your own protocol. Several plug-ins can be loaded in one server instance. You can still have the web server running in parallel with the plug-in. It is also possible to turn of the web server and only run your protocol. That is done by specifying the configuration `HTTP_PORT = 0`.

## 13 SDK

When building plug-ins you can make use of some functions that helps integrating with the server. MinaliC exports a library (minalic.lib) that it is possible to link your plug-ins towards. These functions are:

```
/*Performs logging*/
void echo(char *txt);

/*Performs logging of certain level*/
void echoex(char *txt,int level);

/*Reads parameters in the configuration file*/
char *get_config_value(char *key, char *value);
```

The functions are declared in the source file `minalic.h`. You need to include this file in you project to make use of the functions.

## 14 The plain source code

This section contains the whole source code for the web server. Each source file is headed with its name.

### main.c

```
/*
*****
/*
/*      Minalic Webserver
/*
/*
/*      The minimal webserver developed in C
/*      Author: Hans Alshoff, Sweden 2010
/*
/*
/*      Features:
/*      CGI, PHP and Perl scripts,
/*      directory browsing,
/*      custom plugins
/*
/*      The source code is built in an object
/*      oriented manner.
/*
/*
*****
/*
/*  Development log:
/*  Date      Ver   Build   Desc
/*  2010-08-21  1.0   A001    First version.
/*
/*
/*  This is the main file of the source code.
*/
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include "config.h"
#include "acceptor.h"
#include "acceptorplugin.h"
#include "common.h"
#include "logger.h"
#include "sessionhandler.h"
#include "cgihandler.h"
#include "clientcounter.h"

/*
/*  System global objects and variables.
*/
/* Global path to the server executable*/
char g_module_path[256];
/* Global logger object*/
Logger *g_logger;
/* Global Sessionhandler object*/
Sessionhandler *g_sessionhandler;
/* Global Cgihandler object*/
Cgihandler *g_cgihandler;
/* Global variable holding TCP port number to listen to*/
int g_conf_port;
/* Global variable holding maximum number of clienthandler threads*/
int g_conf_threads;
/* Global variable showing if directory browsing is allowed*/
int g_conf_brows_default;
/* Config file object*/
Config *g_conf;
/* Clientcounter*/
Clientcounter *g_clientcounter;
/* Hexadecimal constants for urlencoding*/
int g_hex_value[256];

/* Global variable holding maximum number of clienthandler threads*/
extern int g_conf_threads;
```

## main.c (continued)

```
/*
 * File global define constants.
 */
#define SERVER_UPDATE_TIME 5000
#define CTIMEOUT 10000

/*
 * File global variables.
 */
volatile static int g_shut_down = 0;
SERVICE_STATUS static g_service_status;
SERVICE_STATUS_HANDLE static g_service_status_handle;
char static g_service_name[100];

/*
 * Function predeclarations.
 */
void WINAPI service_main(DWORD argc, LPSTR argv[]);
void WINAPI server_ctrl_handler(DWORD control);
void update_status(int new_status);
int service_specific(int argc, char *argv[]);

/*****
 * Exported API functions for use from plugins
 */
_declspec(dllexport) void echo(char *txt)
{
    logger_print(g_logger, 1, txt);
}

_declspec(dllexport) void echoex(char *txt, int level)
{
    logger_print(g_logger, level, txt);
}

_declspec(dllexport) char *get_config_value(char *key, char *value)
{
    char *p;

    p = config_getval(g_conf, key);
    if (p)
    {
        strcpy(value, p);
        return value;
    }
    return 0;
}

/*****
 * Start of main code definitions, install functions and
 * service handlers.
 */

/* Function that initializes urlencode hex constants*/
static void setup_hex_constants() {
    int i;
    for (i=0; (i < 256); i++) {
        g_hex_value[i] = 0;
    }
    g_hex_value['0'] = 0;
    g_hex_value['1'] = 1;
    g_hex_value['2'] = 2;
    g_hex_value['3'] = 3;
    g_hex_value['4'] = 4;
    g_hex_value['5'] = 5;
    g_hex_value['6'] = 6;
    g_hex_value['7'] = 7;
    g_hex_value['8'] = 8;
    g_hex_value['9'] = 9;
    g_hex_value['A'] = 10;
    g_hex_value['B'] = 11;
    g_hex_value['C'] = 12;
    g_hex_value['D'] = 13;
    g_hex_value['E'] = 14;
    g_hex_value['F'] = 15;
    g_hex_value['a'] = 10;
    g_hex_value['b'] = 11;
    g_hex_value['c'] = 12;
    g_hex_value['d'] = 13;
```



```

    g_hex_value['e'] = 14;
main.c (continued)

    g_hex_value['f'] = 15;
}

/*
 * Function for creating and initializing server objects.
 */
void initialize_server(Runmode run_mode)
{
    char    configfile_name[FILE_SIZE];
    char    logfile_name[FILE_SIZE];
    size_t  len;
    char    *port;
    char    *log_level;
    int     conflog_level;
    char    *timeout;
    int     conf_timeout;
    char    *tmp;
    char    *threads;
    char    *cgi_timeout;
    int     conf_cgi_timeout;
    char    *brows;

    /* Initialize hex constants*/
    setup_hex_constants();

    /* Get the physical path for the server executable*/
    GetModuleFileNameA(NULL,g_module_path,255);
    len = strlen(g_module_path)-1;
    for (tmp = &g_module_path[len];*tmp != '\\'; tmp--);
    tmp++;
    *tmp = 0; /* Terminate string*/

    /* Create config file path*/
    strcpy(configfile_name,g_module_path);
    strcat(configfile_name,"minalic.ini");

    /*Read config file*/
    g_conf = config_create(configfile_name);

    /* PORT*/
    port = config_getval(g_conf,"HTTP_PORT");
    if (port == NULL)
        g_conf_port = 80; /* No web server port*/
    else
        g_conf_port = atoi(port);

    /* LOGLEVEL*/
    log_level = config_getval(g_conf,"SERVER_LOGLEVEL");
    if (log_level == NULL)
        conflog_level = 1; /* Default value*/
    else
        conflog_level = atoi(log_level);

    /* TIMEOUT*/
    timeout = config_getval(g_conf,"HTTP_SESSION_TIMEOUT");
    if (timeout == NULL)
        conf_timeout = 1800; /* Default value*/
    else
        conf_timeout = atoi(timeout);

    /* CGI TIMEOUT*/
    cgi_timeout = config_getval(g_conf,"CGI_TIMEOUT");
    if (cgi_timeout == NULL)
        conf_cgi_timeout = 90; /* Default value*/
    else
        conf_cgi_timeout = atoi(cgi_timeout);

    /* THREADS*/
    threads = config_getval(g_conf,"SERVER_HANDLERS");
    if (threads == NULL)
        g_conf_threads = 100; /* Default value*/
    else
        g_conf_threads = atoi(threads);

    /* DIRECTORY_BROWSING*/
    brows = config_getval(g_conf,"DIRECTORY_BROWSING_DEFAULT");
    if (brows == NULL)
        g_conf_brows_default = 0; /* Default value*/
    else

```

```

    g_conf_brows_default = atoi(brows);
main.c (continued)

    /* Create log file path*/
    strcpy(logfile_name,g_module_path);
    strcat(logfile_name,"\\minalic.log");

    /* Create global logger object*/
    g_logger = logger_create(logfile_name,run_mode,conflog_level);
    assert(g_logger);

    /* Create global Sessionhandler object*/
    g_sessionhandler = sessionhandler_create(conf_timeout);
    assert(g_sessionhandler);

    /* Create global Cgihandler object*/
    g_cgihandler = cgihandler_create(conf_cgi_timeout);
    assert(g_cgihandler);

    /* Create global Clientcounter object*/
    g_clientcounter = clientcounter_create(g_conf_threads);
    assert(g_clientcounter);

    /* Create the server wwwroot path by adding wwwroot to the server path*/
    strcat(g_module_path,"wwwroot");

    /* Do startup loggings*/
    logger_print(g_logger,1,"MinaliC server %s build:%s",VERSION,BUILD);
    logger_print(g_logger,1,"Server loglevel: %d",conflog_level);
    logger_print(g_logger,1,"Server handlers: %d",g_conf_threads);
    /*Only show webserver values if we have a web server running*/
    if (g_conf_port != 0)
    {
        logger_print(g_logger,1,"Web root: %s",g_module_path);
        logger_print(g_logger,1,"Http session timeout: %d",conf_timeout);
        logger_print(g_logger,1,"CGI timeout: %d",conf_cgi_timeout);
        logger_print(g_logger,1,"Directory browsing: %s",g_conf_brows_default?"ON":"OFF");
    }

#ifdef NDEBUG /* If build in debug*/
    /* Show that debug mode is active*/
    logger_print(g_logger,1,"Debugmode: active");
    //assert(0); /* Dummy assert to se that assertions is active*/
#endif /*NDEBUG*/
}

/*
 * Function for starting the a socket listener thread.
 */
void start_bind(int port,char *plugin_name)
{
    struct sockaddr_in  srv_s_addr;
    SOCKET              srv_sock;
    WORD                 version_requested;
    WSADATA              data;
    int                  err;
    Acceptor             *acceptor = NULL;
    Acceptorplugin       *acceptor_plugin = NULL;

    /* Initialize socket api*/
    version_requested = MAKEWORD(2,0);
    err = WSASStartup(version_requested,&data);
    if (err != 0)
    {
        logger_print(g_logger,1,"Could not find a usable WinSock DLL.");
        exit(1);
    }
    srv_sock = socket(AF_INET,SOCK_STREAM,0);
    srv_s_addr.sin_family = AF_INET;
    /* Accept all addresses*/
    srv_s_addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    /* Port to listen on*/
    srv_s_addr.sin_port = htons(port);

    /* Socket bind to port*/
    if (bind(srv_sock,(struct sockaddr*) &srv_s_addr,sizeof(srv_s_addr)))
    {
        logger_print(g_logger,1,"Could not bind to port %d",port);
        exit(1);
    }
    else
    {

```

## main.c (continued)

```
    /*Check if this is a webserver binding or a custom plugin binding*/
    if (plugin_name == NULL)
    {
        logger_print(g_logger,1,"Web server binding to port: %d, ready for requests",port);
    }
    else
    {
        logger_print(g_logger,1,"Plugin %s binding to port: %d ",plugin_name,port);
    }
}

/* Socket listen on port*/
listen(srv_sock,5);

if (plugin_name == NULL)
{
    /* Create an web acceptor object that performs the socket accepts from clients*/
    acceptor = acceptor_create(srv_sock);
    assert(acceptor);
}
else
{
    /* Create a plugin acceptor object that performs the socket accepts from clients*/
    acceptor_plugin = acceptorplugin_create(srv_sock,plugin_name);
    assert(acceptor_plugin);
}
}

void start_server()
{
    Map          *plugin_map;
    Mapelement  *e;
    int          plugin_port_int;
    char         *plugin_port;
    char         *plugin_module;

    if (g_conf_port != 0)
    {
        /*Start web server socket*/
        start_bind(g_conf_port,NULL);
    }

    /*Start other plugin sockets*/
    plugin_map = config_get_plugins(g_conf);
    e = map_findfirst(plugin_map);
    if (e != NULL)
    {
        do
        {
            plugin_module = e->key;
            plugin_port = e->value;
            plugin_port_int = atoi(plugin_port);
            start_bind(plugin_port_int,plugin_module);
            e = map_findnext(plugin_map);
        }while(e != NULL);
    }

    /* Delete the Config object since we don't need it anymore*/
    config_delete(g_conf);
}

/*
 * Function that cleans up server objects and close the socket api.
 */
void endServer()
{
    sessionhandler_delete(g_sessionhandler);
    cgihandler_delete(g_cgihandler);
    clientcounter_delete(g_clientcounter);
    logger_delete(g_logger);
    WSACleanup();
}

/*
 * Function for showing the help text
 */
void ShowUsage()
{
    printf("minalic -install | -stop | -start | -uninstall | -run\n");
}
```

## main.c (continued)

```
/*
 * Function that performs the windows service installation.
 */
Result DoInstallService(int argc, char *argv [])
{
    SC_HANDLE    service;
    SC_HANDLE    sc_manager;
    char         executable[256];
    char         exec_path[256]="";

    strcpy(g_service_name,"Minalic webserver");
    if (argc == 1)
    {
        if (!_strnicmp(argv[0],"-name:",6)) /*called by user to install/uninstall*/
        {
            strcpy(g_service_name,&argv[0][6]);
        }
    }

    sc_manager = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
    GetModuleFileNameA(NULL,executable,256);
    strcat(executable," -srv");

    if ( sc_manager )
    {
        service = CreateServiceA(sc_manager,
                                (LPCSTR)g_service_name,
                                (LPCSTR)g_service_name,
                                SERVICE_START | SERVICE_STOP,
                                SERVICE_WIN32_OWN_PROCESS,
                                SERVICE_AUTO_START,
                                SERVICE_ERROR_NORMAL,
                                executable,
                                NULL,NULL,NULL,NULL,NULL);

        if (service)
        {
            StartService(service,0,NULL);
            CloseServiceHandle(service);
            return OK;
        }
    }
    return ERR;
}

/*
 * Function that stops and uninstalls the windows service.
 */
Result DoRemoveService(int argc, char *argv [])
{
    SC_HANDLE    service;
    SC_HANDLE    sc_manager;
    Result       res = ERR;

    strcpy(g_service_name,"Minalic webserver");
    if (argc == 1)
    {
        if (!_strnicmp(argv[0],"-name:",6)) /*called by user to install/uninstall*/
        {
            strcpy(g_service_name,&argv[0][6]);
        }
    }

    sc_manager = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
    if ( sc_manager )
    {
        service = OpenServiceA(sc_manager,(LPCSTR)g_service_name,SERVICE_ALL_ACCESS);

        if (service)
        {
            /*Try to stop the service*/
            if (ControlService(service,SERVICE_CONTROL_STOP,&g_service_status))
            {
                printf("Stopping %s.", g_service_name);
                Sleep(1000);

                while(QueryServiceStatus(service,&g_service_status))
                {
                    if (g_service_status.dwCurrentState == SERVICE_STOP_PENDING)
                    {

```

## main.c (continued)

```
        printf(".");
        Sleep(1000);
    }
    else
        break;
}

if (g_service_status.dwCurrentState == SERVICE_STOPPED)
{
    printf("\n stopped.\n");
}
else
    printf("\n failed to stop.\n");
}

/* Now remove the service*/
if(DeleteService(service))
{
    printf("%s removed.\n",g_service_name);
    res = OK;
}
CloseServiceHandle(service);
}
CloseServiceHandle(sc_manager);
}
return res;
}

/*
 * Function that stops the windows service.
 */
Result DoStopService(int argc, char *argv [])
{
    SC_HANDLE    service;
    SC_HANDLE    sc_manager;
    Result        res = ERR;

    strcpy(g_service_name,"Minalic webserver");
    if (argc == 1)
    {
        if (!_strnicmp(argv[0],"-name:",6)) /*called by user to install/uninstall*/
        {
            strcpy(g_service_name,&argv[0][6]);
        }
    }

    sc_manager = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
    if (sc_manager)
    {
        service = OpenServiceA(sc_manager,(LPCSTR)g_service_name,SERVICE_ALL_ACCESS);

        if (service)
        {
            /* Try to stop the service*/
            if (ControlService(service,SERVICE_CONTROL_STOP,&g_service_status))
            {
                printf("Stopping %s.", g_service_name);
                Sleep(1000);

                while(QueryServiceStatus(service,&g_service_status))
                {
                    if (g_service_status.dwCurrentState == SERVICE_STOP_PENDING)
                    {
                        printf(".");
                        Sleep(1000);
                    }
                    else
                    {
                        break;
                    }
                }

                if (g_service_status.dwCurrentState == SERVICE_STOPPED)
                {
                    printf("\n stopped.\n");
                    res = OK;
                }
                else
                    printf("\n failed to stop.\n");
            }
        }
    }
}
```

## main.c (continued)

```
        CloseServiceHandle(service);
    }
    CloseServiceHandle(sc_manager);
}
return res;
}

/*
 * Function that starts the windows service.
 */
Result DoStartService(int argc, char *argv [])
{
    SC_HANDLE    service;
    SC_HANDLE    sc_manager;
    Result        res = ERR;

    strcpy(g_service_name,"Minalic webserver");
    if (argc == 1)
    {
        if (!_strnicmp(argv[0],"-name:",6)) /*called by user to install/uninstall*/
        {
            strcpy(g_service_name,&argv[0][6]);
        }
    }

    sc_manager = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
    if (sc_manager)
    {
        service = OpenServiceA(sc_manager,(LPCSTR)g_service_name,SERVICE_ALL_ACCESS);

        if (service)
        {
            StartService(service,0,NULL);
            CloseServiceHandle(service);
            res = OK;
        }
        CloseServiceHandle(sc_manager);
    }
    return res;
}

/*
 * The main function of the server process.
 * This function acts both as the main install function called as a normal application
 * and as as a routine that starts the service control dispatcher from SCM (Service Control Manager).
 */
void main(int argc, char *argv [])
{
    if (argc<2) /* Started as an application*/
    {
        /* Initialized server to start as an application*/
        initialize_server(APPLICATION);
        start_server();
        /* Main loop for the server*/
        do
        {
            Sleep (SERVER_UPDATE_TIME);
            /* Periodically delete old sessions in the Sessionhandler*/
            sessionhandler_delete_old(g_sessionhandler);
            /* Periodically delete old cgi processes*/
            cgihandler_delete_old(g_cgihandler);
        }while(1);
        endServer();
    }
    else
    {
        printf("MinaliC server %s build:%s Command Line\n",VERSION,BUILD);
        /*The servers is started as an application*/
        if (!_strnicmp(argv[1],"-uninstall",10)) /*called by user to install/uninstall*/
        {
            if (DoRemoveService(argc-2,&argv[2]) == OK)
                printf("Command completed successfully");
        }
        else if (!_strnicmp(argv[1],"-stop",5)) /*called by user to install/uninstall*/
        {
            if (DoStopService(argc-2,&argv[2]) == OK)
                printf("Command completed successfully");
        }
        else if (!_strnicmp(argv[1],"-start",6)) /*called by user to install/uninstall*/
        {

```

## main.c (continued)

```
        if (DoStartService(argc-2,&argv[2]) == OK)
            printf("Command completed successfully");
    }
    else if (!_stricmp(argv[1],"-install"))
    {
        if (DoInstallService(argc-2,&argv[2]) == OK)
            printf("Command completed successfully");
    }
    else if (!_strnicmp(argv[1],"-",2))
    {
        ShowUsage();
    }
    else if (!_strnicmp(argv[1],"-srv",2)) /* Started from service control managed*/
    {
        SERVICE_TABLE_ENTRY DispatchTable[] =
        {
            { g_service_name,          service_main},
            { NULL,                     NULL }
        };

        /*The servers is initialized to start as a service*/
        initialize_server(SERVICE);

        if (!StartServiceCtrlDispatcherA(DispatchTable))
        {
            ShowUsage();
        }
    }
    else
    {
        printf("Error - wrong number of arguments\n\n");
        ShowUsage();
    }
}
return;
}

/*
 * Service entry point, called when the service is created.
 */
void WINAPI service_main(DWORD argc, LPSTR argv[])
{
    /* The service runs in its own process*/
    g_service_status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    /* The service is starting*/
    g_service_status.dwCurrentState = SERVICE_START_PENDING;
    /* Can be stopped and be notified when system is shut_down*/
    g_service_status.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN;
    /* Running normal (no error)*/
    g_service_status.dwWin32ExitCode = NO_ERROR;
    /* Running normal (no error)*/
    g_service_status.dwServiceSpecificExitCode = NO_ERROR;
    /* Service incremental checkpoint reset*/
    g_service_status.dwCheckPoint = 0;
    /* An estimate of the amount of time needed in pending modes*/
    g_service_status.dwWaitHint = 2*CSTIMEOUT;

    /* Register a function to handle the service control requests*/
    g_service_status_handle = RegisterServiceCtrlHandlerA((LPCSTR)g_service_name,server_ctrl_handler);
    if (g_service_status_handle == 0)
        logger_print(g_logger,1,"Cannot register control handler");

    /* Updates the service control manager's status */
    SetServiceStatus(g_service_status_handle,&g_service_status);

    /* Start the service-specific work, now when the generic work is complete */
    if (service_specific(argc,argv) != 0)
    {
        g_service_status.dwCurrentState = SERVICE_STOPPED;
        g_service_status.dwServiceSpecificExitCode = 1; /* Server initialization failed */
        SetServiceStatus (g_service_status_handle, &g_service_status);
        return;
    }

    /* Here the serverspecific execution is finished, so report status stopped*/
    update_status(SERVICE_STOPPED);

    return;
}
```

## main.c (continued)

```
/*
 * This is the service specific function called when the service is started.
 */
int service_specific(int argc, char *argv[])
{
    /* At this point we are running so change to status running*/
    update_status(SERVICE_RUNNING);

    /* Start the webserver*/
    start_server();

    /* Main loop for service waiting for shut_down*/
    while (!g_shut_down)
    {
        /* Shut_down is set on a shut down control in SCM*/
        Sleep(SERVER_UPDATE_TIME);
        /* Periodically delete old sessions in the Sessionhandler*/
        sessionhandler_delete_old(g_sessionhandler);
        /* Periodically delete old cgi processes*/
        cgihandler_delete_old(g_cgihandler);
    }
    logger_print(g_logger, 1, "Server stopped.");
    endServer();
    return 0;
}

/*
 * Callback control handler called from SCM to change status of the service.
 */
void WINAPI server_ctrl_handler(DWORD control)
{
    switch (control)
    {
        case SERVICE_CONTROL_SHUTDOWN:
        case SERVICE_CONTROL_STOP:
            /* Answer SCM with stop_pending*/
            update_status(SERVICE_STOP_PENDING);
            /* Set the global shut_down flag */
            g_shut_down = 1;
            break;
        case SERVICE_CONTROL_PAUSE:
            break;
        case SERVICE_CONTROL_CONTINUE:
            break;
        case SERVICE_CONTROL_INTERROGATE:
            break;
        default:
            if (control > 127 && control < 256) /* User Defined */
                break;
    }
    return;
}

/*
 * Set a new status and checkpoint for the service.
 */
void update_status(int new_status)
{
    if (new_status >= 0) g_service_status.dwCurrentState = new_status;
    if (!SetServiceStatus(g_service_status_handle, &g_service_status))
        logger_print(g_logger, 1, "Failed to set status for service", g_service_name);
    return;
}
```



## common.h

```
/*
 * Constant values used in the server.
 */

#ifndef _COMMON
#define _COMMON

/* Server version presented at startup*/
#define VERSION "1.0.0"
/* Server build version presented at startup*/
#define BUILD "A001"
/* Max size of a http request header*/
#define HEADER_SIZE 2048
/* Maximum size of the http request method (GET,POST)*/
#define METHOD_SIZE 20
/* Maximum size of a phisical file and path name*/
#define FILE_SIZE 256
/* Maximum size of the querystring in a http request*/
#define QUERY_SIZE 256
/* Maximum size of the version part in a http request*/
#define VERSION_SIZE 20
/* Maximum size of the phrase part in a http request*/
#define PHRASE_SIZE 40
/* Max size of the status code part in a http response*/
#define STATUS_SIZE 5
/* Maximun size of host header*/
#define HOST_SIZE 256
/* Maximum size of the cookie header in a http response*/
#define COOKIE_SIZE (HOST_SIZE + 50)
/* The size of a session id string*/
#define SESSION_ID_SIZE 31
/* Maximum number of retries to create a unique session id*/
#define SESSION_MAX_TRY 50
/* The size of the chunk to increase the cgi response buffer with*/
#define CGI_BUFFER_SIZE 1024
/* The size of the chunk to increase the directory browsing response buffer with*/
#define DIR_BUFFER_SIZE 1024
/* The maximum size of the headers returned from a cgi script*/
#define CGI_HEADER_MAX 512
/* The maximum size of a file extension*/
#define FILE_EXTENSION 10
/* The maxumum number of internal http redirections*/
#define RECURSIVE_CALLS_ALLOWED 10
/* The size of the log buffer*/
#define LOG_FORMAT_SIZE 200
/* The size of the http Date header*/
#define HTTP_TIME_SIZE 80
/* Maximum size of the Content-Type header*/
#define CONTENT_TYPE_SIZE 100
/* The size of the chunk to send a file with*/
#define SEND_FILE_CONTENT_CHUNK_SIZE 256
/* Server script default file strings*/
#define BINARY_DEFAULT_FILE "index.exe"
#define CGI_DEFAULT_FILE "index.cgi"
#define PHP_DEFAULT_FILE "index.php"
#define PERL_DEFAULT_FILE "index.pl"
#define HTM_DEFAULT_FILE "index.htm"

/* The cookie name of the session id cookie*/
#define SESSION_ID_COOKIE_NAME "session_id"

/* Name of directory browsing file*/
#define DIRECTORY_BROWSING_FILE "dir.ini"

/* Enum constants used as return value from functions*/
typedef enum Result
{
    OK=0,          /* Success*/
    ERR=1,         /* Error*/
    ENDED=2,       /* Operation is finished, don't proceed*/
    NOT_FOUND=3    /* Page is not found*/
} Result;

/* Enum constants used to specify run mode for the server*/
typedef enum Runmode
{
    SERVICE=0,     /* Run as service*/
    APPLICATION=1 /* Run as application*/
} Runmode;
#endif /* _COMMON*/
```

## config.h

```
/*
 * Declaration of the Config class.
 * This class is a helper class to read from
 * the configuration file.
 */
#define _CRT_SECURE_NO_WARNINGS
#ifndef _CONFIG
#define _CONFIG

#include "common.h"
#include "map.h"

/*
 * The Config data type.
 */
typedef struct Config
{
    char _file[FILE_SIZE];
    Map *_values;
    Map *_plugin_values;
} Config;

/* The Config methods*/
Config *config_create(char *file_name);
void config_delete(Config *self);
char *config_getval(Config *self, char *varname);
Map *config_get_plugins(Config *self);

#endif /* _CONFIG*/
```

## config.c

```
/*
 * Definition of the Config class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "config.h"
#include "logger.h"

/* Define maximum length of a row in the config file*/
#define CONFIG_LINE 400

/*
 * Private method that parses a config line.
 * If it finds a proper config variable it will
 * be added to the Map that holds all variables.
 */
static void parseLine(Config *self, char *line)
{
    char *p;
    char *key;
    char *value;
    char *value2;

    p = line;
    while(*p == ' ' || *p == '\t')
    {
        p++;
        if (!*p)
            return;
    }

    key = p;

    while(*p != ' ' && *p != '=')
    {
        p++;
        if (!*p)
            return;
    }

    *p = 0;

    p++;
    if (*p)
    {
        while(*p == ' ' || *p == '=' || *p == '\t')
        {
            p++;
            if (!*p)
                return;
        }
    }

    value = p;

    while(*p != ' ' && *p != '\n' && *p != 0)
    {
        p++;
    }

    *p = 0;

    if (!_stricmp(key, "SERVER_PLUGIN"))
    {
        for (value2 = value; *value2 != ':'; value2++);
        *value2 = 0;
        value2++;
        /* Add plugin name and portnumber as a pair into the plugin map*/
        map_set(self->_plugin_values, value, value2);
    }
    else
    {
        /* Add found variable and its value to the Map*/
        map_set(self->_values, key, value);
    }
}
```

## config.c (continued)

```
/*
 * Private method that reads the config file.
 */
static void readFile(Config *self)
{
    FILE *f = NULL;
    char line[CONFIG_LINE+1];

    f = fopen(self->_file, "r");
    if (!f)
        return;

    while(!feof(f))
    {
        fgets(line, CONFIG_LINE, f);
        if (line[0] != '*')
        {
            parseLine(self, line);
        }
    }
    fclose(f);
}

/*
 * Constructor
 */
Config *config_create(char *file_name)
{
    Config *new_obj;

    new_obj = malloc(sizeof(Config));
    assert(new_obj);
    if (new_obj)
    {
        strcpy(new_obj->_file, file_name);
        new_obj->_values = map_create();
        new_obj->_plugin_values = map_create();
        readFile(new_obj);
    }
    return new_obj;
}

/*
 * Destructor
 */
void config_delete(Config *self)
{
    assert(self);
    if (self)
    {
        map_delete(self->_values);
        map_delete(self->_plugin_values);
        free(self);
    }
}

/*
 * Method that returns the value of a config variable.
 */
char *config_getval(Config *self, char *varname)
{
    assert(self);
    assert(varname);
    return map_getval(self->_values, varname);
}

/*
 * Method that returns the plugin map.
 */
Map *config_get_plugins(Config *self)
{
    assert(self);
    return self->_plugin_values;
}
```

## logger.h

```
/*
 * Declaration of the Logger class.
 * This class handles all logging to file
 * and stdout.
 */

#ifndef _LOGGER
#define _LOGGER

#include <windows.h>
#include "common.h"

/*Definition of the Logger type*/
typedef struct Logger
{
    char    _file[FILE_SIZE];
    Runmode _Runmode;
    int     _log_level;
    /* Logger is used by several threads and need a critical section*/
    CRITICAL_SECTION _critical_section;
} Logger;

/* The Logger methods*/
Logger *logger_create(char *file_name,Runmode mode,int log_level);
void logger_delete(Logger *self);
void logger_print(Logger *self,int level,char *format,...);

/*Declaration of global logger instance*/
extern Logger *g_logger;

/* The assert declaration*/
#undef assert
#ifdef NDEBUG
/*Declaration of assert in relese mode*/
#define assert(Expression) ((void)0)
#else
/* Declaration of global logger object needed in assert*/
/* Declaration of the assert macro used to log failure in debug mode*/
#define assert(Expression) (!(Expression) ? logger_print(g_logger,1,"Fatal error %s %s\n",#Expression, __FILE__ , __LINE__ ):(void)0)
#endif /* NDEBUG */

#endif /*_LOGGER*/
```

## logger.c

```
/*
 * Definition of the Logger class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdarg.h>
#include "logger.h"

/*
 * Constructor
 */
Logger *logger_create(char *file_name, Runmode mode, int log_level)
{
    Logger *new_obj;
    new_obj = malloc(sizeof(Logger));
    assert(new_obj);
    if (new_obj)
    {
        strcpy(new_obj->_file, file_name);
        new_obj->_log_level = log_level;
        new_obj->_Runmode = mode;
        InitializeCriticalSection(&new_obj->_critical_section);
    }
    return new_obj;
}

/*
 * Destructor
 */
void logger_delete(Logger *self)
{
    assert(self);
    DeleteCriticalSection(&self->_critical_section);
    free(self);
}

/*
 * Method that prints to the log.
 * The log can be the log file or standad output.
 * The method takes a format string and a variable number of parameters.
 */
void logger_print(Logger *self, int level, char *format, ...)
{
    FILE *out;
    time_t rawtime;
    struct tm *timeinfo;
    char log_time[50];
    char log_format[LOG_FORMAT_SIZE];
    va_list args;

    /*
     * If the level of the message is higher than the current
     * loglevel don't log.
     */
    if (level > self->_log_level)
        return;

    if (self == NULL)
        return;

    /*
     * Protect this section by a CriticalSection object so it not will be used
     * by several threads at the same time.
     */
    EnterCriticalSection(&self->_critical_section);
    out = NULL;
    if (self->_Runmode == SERVICE)
        out = fopen(self->_file, "a");
    else if (self->_Runmode == APPLICATION)
        out = stdout;

    if (!out)
    {
        LeaveCriticalSection(&self->_critical_section);
        return;
    }

    time ( &rawtime );
```

## logger.c (continued)

```
    timeinfo = localtime( &rawtime );
    strftime(log_time,sizeof(log_time),"%Y-%b-%d %H:%M:%S",timeinfo);
    va_start(args, format);
    sprintf(log_format,"%s | %s\n",log_time,format);
    vfprintf(out,log_format,args);

    if (self->_Runmode == SERVICE)
        fclose(out);

    va_end(args);
    LeaveCriticalSection(&self->_critical_section);
    /* End of critical section*/
}
```

## acceptor.h

```
/*
 * Declaration of the Acceptor class.
 * This class is responsible for connecting the
 * clients to the sever.
 */

#ifndef _ACCEPTOR
#define _ACCEPTOR

#include "thread.h"

/*
 * The Acceptor data type.
 * Acceptor is a thread class that includes
 * a Thread instance.
 */
typedef struct Acceptor
{
    /* Acceptor is a thread so include a thread instance*/
    Thread      _base_thread;
    SOCKET      _srv_socket;
    SOCKET      _sockio;
    struct sockaddr_in _addr;
    int         _addr_len;
} Acceptor;

/* The Acceptor methods*/
Acceptor *acceptor_create(SOCKET s);
Acceptor *acceptor_create_plugin(SOCKET s);
void acceptor_delete(Acceptor *self);
static void acceptor_run(void* self);
static void acceptor_run_plugin(void* self);

#endif /*_ACCEPTOR*/
```



## acceptor.c

```
/*
 * Definition of the Acceptor class
 */
#define _CRT_SECURE_NO_WARNINGS
#include "acceptor.h"
#include "clienthandler.h"
#include "logger.h"

/* Declaration of global logger object*/

/*
 * Constructor
 */
Acceptor *acceptor_create(SOCKET s)
{
    Acceptor *new_obj;

    new_obj = malloc(sizeof(Acceptor));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_base_thread._run = acceptor_run;
        new_obj->_base_thread._delete = acceptor_delete;
        new_obj->_srv_socket = s;
        new_obj->_addr_len = sizeof(new_obj->_addr);
        thread_start(&new_obj->_base_thread);
    }
    return new_obj;
}

/*
 * Destructor
 */
void acceptor_delete(Acceptor *self)
{
    assert(self);
    if (self)
    {
        shutdown(self->_sockio, 2);
        closesocket(self->_sockio);
        CloseHandle(self->_base_thread._handle);
        free(self);
    }
}

/*
 * Thread function of Acceptor that performs
 * the real work.
 */
static void acceptor_run(void* self)
{
    Clienthandler *ch;
    Acceptor *a = (Acceptor*)self;

    assert(a);
    if (a)
    {
        while (1)
        {
            /* Wait for a client to connect to the server socket*/
            a->_sockio = accept(a->_srv_socket, (struct sockaddr*)&a->_addr, &a->_addr_len);

            /* When a client connects create a client handler
             * thread and let it handle the request.
             * Let the client handler run for it self. It will be deleted
             * when it ends.
             */
            ch = clienthandler_create(a->_sockio, a->_addr);
            assert(ch);
        } /* Return to the top of the loop and handle the next incoming request*/
    }
}
```

## clienthandler.h

```
/*
 * Declaration of the Clienthandler class.
 * This class handles a client request from start
 * to the end. It runs in a separate thred.
 */

#ifndef _CLIENTHANDLER
#define _CLIENTHANDLER

#include "thread.h"
#include "common.h"
#include "request.h"
#include "response.h"

/*
 * The Clienthandler data type.
 * Clienthandler is a thread class that includes
 * a Thread instance.
 */
typedef struct Clienthandler
{
    /* Clienthandler is a thread so include a thread instance*/
    Thread    _base_thread;
    SOCKET    _sockio;
    struct    sockaddr_in _addr;
    Request   *_request;
    Response  *_response;
} Clienthandler;

/* The Clienthandler methods*/
Clienthandler *clienthandler_create(SOCKET s,struct    sockaddr_in addr);
void clienthandler_delete(Clienthandler *self);
static void clienthandler_run(void* self);

#endif /*_CLIENTHANDLER*/
```

## clienthandler.c

```
/*
 * Definition of the Clienthandler class
 */
#define _CRT_SECURE_NO_WARNINGS
#include "clienthandler.h"
#include "request.h"
#include "response.h"
#include "logger.h"
#include "clientcounter.h"

/* Global Clientcounter object*/
extern Clientcounter *g_clientcounter;

/*
 * Constructor
 */
Clienthandler *clienthandler_create(SOCKET s, struct sockaddr_in addr)
{
    Clienthandler *new_obj = NULL;

    /*Check that we dont' create to many simultaneous threads*/
    if (clientcounter_hold_client(g_clientcounter))
    {
        /* Allocate memory for this object*/
        new_obj = malloc(sizeof(Clienthandler));
        assert(new_obj);
        if (new_obj)
        {
            new_obj->_base_thread._run = clienthandler_run;
            /*Set virtual destructor to the destructor of this class*/
            new_obj->_base_thread._delete = clienthandler_delete;
            new_obj->_sockio = s;
            new_obj->_addr = addr;
            /*Create a Request object for the client request*/
            new_obj->_request = request_create(new_obj->_sockio, addr);
            /*Create a Response object for the client response*/
            new_obj->_response = response_create(new_obj->_sockio);
            /*Start the work of the thread for this object*/
            thread_start(&new_obj->_base_thread);
        }
    }

    /*If failed to create a clienthandler we close the socket*/
    if (new_obj == NULL)
    {
        /* Close the socket for this object*/
        shutdown (s,2);
        closesocket(s);
    }
    return new_obj;
}

/*
 * Destructor
 */
void clienthandler_delete(Clienthandler *self)
{
    assert(self);
    if (self)
    {
        /* Close the socket for this object*/
        shutdown (self->_sockio,2);
        closesocket(self->_sockio);
        /* Close the handle to the thread*/
        CloseHandle(self->_base_thread._handle);
        /* Delete the referensig objects*/
        request_delete(self->_request);
        response_delete(self->_response);
        /* Free memory for this object*/
        free(self);
        /*Count down number of simultanous clienthandler threads*/
        clientcounter_release_client(g_clientcounter);
    }
}
```

## clienthandler.c (continued)

```
/*
 * Thread function of Clienthandler
 */
static void clienthandler_run(void* self)
{
    Clienthandler *me = (Clienthandler*)self;
    Result res;

    assert(me);
    /* Try to read a http header from the request*/
    if (request_read_header(me->_request) == OK)
    {
        /*
         * Check if http method is GET or POST
         * which are the only two methods implemented by the server.
         */
        if ( (_strcmp(me->_request->_method,"GET") == 0) ||
            (_strcmp(me->_request->_method,"POST") == 0) )
        {
            /*
             * Handle GET and POST the same way by assuming
             * that some kind of file is requested.
             */
            res = response_send_file(me->_response,me->_request);

            /* Check the result from the Response object*/
            if ( res != OK)
            {
                switch(res)
                {
                    case NOT_FOUND:
                        /* Requested file doesn't exist*/
                        response_send_not_found(me->_response);
                        break;
                    default:
                        /*
                         * Send 500 'Internal server error'
                         * for all other errors.
                         */
                        response_send_internal_server_error(me->_response);
                }
            }
        }
        else
        {
            /*
             * Send 501 'Not Implemented' if request method
             * was not GET or POST.
             */
            response_send_not_implemented(me->_response);
        }
    }
    else
    {
        /*
         * Send 400 'Bad Request' if failed to
         * read a proper http header.
         */
        response_send_bad_request(me->_response);
    }
}
```

## thread.h

```
/*
 * Declaration of the Thread class.
 * This class is a base class for all classes
 * that runs in a separate thread.
 */

#ifndef _THREAD
#define _THREAD

#include <windows.h>

/*
 * The Thread data type.
 */
typedef struct Thread
{
    HANDLE _handle;
    DWORD _id;
    /* Virtual run method*/
    void (*_run)(void* self);
    /* Virtual destructor*/
    void (*_delete)(void* self);
} Thread;

/* The Thread methods*/
Thread *thread_create();
void thread_delete(Thread *self);
void thread_run(void* self);
void thread_start(Thread *self);
int thread_isRunning(Thread *self);
void thread_terminate(Thread *self);
void thread_suspend(Thread *self);
void thread_resume(Thread *self);

#endif /*_THREAD*/
```

## thread.c

```
/*
 * Definition of the Thread class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include "thread.h"
#include "logger.h"

/* Declaration of the thread function to run for all threads*/
unsigned int thread_func (void *self);

/* Declaration of the lib function for creating win32 threads*/
HANDLE _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned ( *start_address )( void * ),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr
);

/*
 * Constructor
 */
Thread *thread_create()
{
    Thread *new_obj;
    new_obj = malloc(sizeof(Thread));
    assert(new_obj);
    if (new_obj)
    {
        /* Set the function to run*/
        new_obj->_run = thread_run;
        /* Set the destructor to run after execution*/
        new_obj->_delete = thread_delete;
        /* Start the threads execution*/
        thread_start(new_obj);
    }
    return new_obj;
}

/*
 * Destructor
 */
void thread_delete(Thread *self)
{
    assert(self);
    assert(self->_handle);
    CloseHandle(self->_handle);
    free(self);
}

/*
 * Method that starts thread execution.
 */
void thread_start(Thread *self)
{
    assert(self);
    self->_handle = (HANDLE) _beginthreadex(NULL, 0, thread_func, self, 0, (LPDWORD)&self->_id);
}

/*
 * Method that suspends thread execution.
 */
void thread_suspend(Thread *self)
{
    assert(self);
    SuspendThread(self->_handle);
}

/*
 * Method that resumes thread execution after suspend.
 */
void thread_resume(Thread *self)
{
    assert(self);
    ResumeThread(self->_handle);
}
```

## thread.c (continued)

```
/*
 * Method that returns if a thread is still running.
 */
int thread_isRunning(Thread *self)
{
    DWORD exit_code;

    assert(self);
    GetExitCodeThread(self->_handle,&exit_code);
    return (exit_code == STILL_ACTIVE);
}

/*
 * Method that forces termination of thread execution.
 */
void thread_terminate(Thread *self)
{
    assert(self);
    TerminateThread(self->_handle,0);
}

/*
 * Static thread function for all thread instances.
 * The function takes as parameter a reference to a thread instance.
 */
static unsigned int thread_func (void *self)
{
    Thread *t;
    t = (Thread*)self;
    assert(t);
    /* Run the threads run function*/
    t->_run(self);
    /*
     * Here the threads job is done and we
     * call the virtual destructor to automaticly
     * clean up memory.
     */
    t->_delete(self);

    return 0;
}

/*
 * Private method that has no implementation due to
 * that the actual instances of the thread class has
 * its own thread functions.
 */
static void thread_run(void* self)
{
}
```

## request.h

```
/*
 * Declaration of the Request class.
 * This class is responsible for reading the socket
 * input stream and interpret it as a http message.
 */

#ifndef _REQUEST
#define _REQUEST

#include "common.h"
#include "map.h"

/*
 * The Request data type.
 */
typedef struct Request
{
    SOCKET    _socket;
    struct    sockaddr_in _addr;
    char      _method[METHOD_SIZE];
    char      _file[FILE_SIZE];
    char      _query[QUERY_SIZE];
    char      _version[VERSION_SIZE];
    int       _contentExist;
    char      _session_id[SESSION_ID_SIZE];
    Map       *_headers;
} Request;

/* The Request methods*/
Request *request_create(SOCKET s, struct sockaddr_in);
void request_delete(Request *self);
Result request_read_header(Request *self);
int request_read_content(Request *self, char *buf, int len);

#endif /* _REQUEST */
```



## request.c

```
/*
 * Definition of the Request class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include "request.h"
#include "logger.h"

/* Use global hexadecimal constants for urlencoding*/
extern int g_hex_value[256];

/* Escape states for urlencoding*/
typedef enum {
    esc_rest,
    esc_first,
    esc_second
} Escstate;

/*
 * Constructor
 */
Request *request_create(SOCKET s, struct sockaddr_in a)
{
    Request *new_obj;

    new_obj = malloc(sizeof(Request));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_socket = s;
        new_obj->_addr = a;
        new_obj->_headers = map_create();
    }

    return new_obj;
}

/*
 * Destructor
 */
void request_delete(Request *self)
{
    assert(self);
    assert(self->_headers);
    map_delete(self->_headers);
    free(self);
}

/*
 * Private helper function that removes double dots and slashes
 */
static void remove_double_dots_and_double_slashes(char *s)
{
    char *p = s;

    while (*s != '\0')
    {
        *p++ = *s++;
        if (s[-1] == '/' || s[-1] == '\\')
        {
            /* Skip all following slashes and backslashes */
            while (*s == '/' || *s == '\\')
            {
                s++;
            }

            /* Skip all double-dots */
            while (*s == '.' && s[1] == '.')
            {
                s += 2;
            }
        }
    }
    *p = '\0';
}
```

## request.c (continued)

```
/* Unescape urlencoded string*/
static int unescape_chars(char *sp, char *cp, int len)
{
    Escstate escape_state = esc_rest;
    int escaped_value = 0;
    int src_pos = 0;
    int dst_pos = 0;

    while (src_pos < len)
    {
        int ch = cp[src_pos];
        switch (escape_state)
        {
            case esc_rest:
                if (ch == '%')
                {
                    escape_state = esc_first;
                }
                else if (ch == '+')
                {
                    sp[dst_pos++] = ' ';
                }
                else
                {
                    sp[dst_pos++] = ch;
                }
                break;
            case esc_first:
                escaped_value = g_hex_value[ch] << 4;
                escape_state = esc_second;
                break;
            case esc_second:
                escaped_value += g_hex_value[ch];
                sp[dst_pos++] = escaped_value;
                escape_state = esc_rest;
                break;
        }
        src_pos++;
    }
    sp[dst_pos] = '\0';

    return 1;
}

/*
 * Private method that parses the http request header.
 * The method looks for the following parts:
 * HTTP_METHOD, URL, QUERY_STRING, HTTP_VERSION
 * It also looks for all http headers that are found.
 */
static Result parse_header(Request *self, char *header)
{
    char *p;
    char *name;
    char *value;
    int index;
    int parsed;
    int end_of_header;
    int has_query;
    int host_found = 0;
    char tmp_file[FILE_SIZE];
    int file_len;

    assert(self);
    assert(header);

    /* HTTP_METHOD*/
    for(parsed = 0, p = header, index = 0; index < METHOD_SIZE - 1; index++, p++)
    {
        if (*p == ' ')
        {
            {
                parsed = 1;
                break;
            }
        }
        if (*p == '\r')
        {
            {
                break;
            }
        }
        self->_method[index] = *p;
    }
}
```

## request.c (continued)

```
self->_method[index] = 0;

if (!parsed)
    return ERR;

/* URL*/
p++;
has_query = 0;
file_len = 0;
for(parsed = 0, index = 0; index < FILE_SIZE - 1; index++, p++)
{
    if (*p == ' ')
    {
        parsed = 1;
        has_query = 0;
        break;
    }
    else if (*p == '?')
    {
        parsed = 1;
        has_query = 1;
        break;
    }
    else if (*p == '\r')
    {
        break;
    }
    tmp_file[index] = *p;
    file_len++;
}
tmp_file[index] = 0;

unescape_chars(self->_file, tmp_file, file_len);
remove_double_dots_and_double_slashes(tmp_file);

if (!parsed)
    return ERR;

if (has_query)
{
    /* QUERY_STRING*/
    p++;
    has_query = 0;
    for(parsed = 0, index = 0; index < QUERY_SIZE - 1; index++, p++)
    {
        if (*p == ' ')
        {
            parsed = 1;
            break;
        }
        else if (*p == '\r')
        {
            break;
        }
        self->_query[index] = *p;
    }
    self->_query[index] = 0;

    if (!parsed)
        return ERR;
}
else
{
    self->_query[0] = 0;
}

/* HTTP_VERSION*/
p++;
for(parsed = 0, index = 0; index < VERSION_SIZE - 1; index++, p++)
{
    if (*p == '\r')
    {
        parsed = 1;
        break;
    }
    self->_version[index] = *p;
}
self->_version[index] = 0;
```

## request.c (continued)

```
/* Parse other headers found*/
if (parsed)
{
    end_of_header = 0;
    do
    {
        p++; /* skip '\n'*/
        p++;
        name = p;
        parsed = 0;
        for(;;p++)
        {
            /*Check that we don't read over header bound*/
            if ( (p - header) >= HEADER_SIZE)
                return ERR;

            if (*p == ' ')
            {
                parsed = 1;
                break;
            }
            if (*p == '\r')
            {
                parsed = 1;
                break;
            }
        }
    }
    *p = 0;
    if (parsed)
    {
        p++;
        value = p;
        for(;*p != '\r';p++)
        {
            /*Check that we don't read over header bound*/
            if ( (p - header) >= HEADER_SIZE)
                return ERR;
        }
        *p = 0;

        if (*(p+2) == '\r')
            end_of_header = 1;

        /*
         * Header name and value found so add it to the header map.
         */
        /* Collect headers that must exist in a proper request*/
        if (!_stricmp(name,"host:"))
        {
            /*Check that host is not already found*/
            if (host_found)
            {
                return ERR;
            }
            /*Check size of host value field*/
            if (strlen(value) <= HOST_SIZE)
            {
                host_found = 1;
            }
        }
        map_add(self->headers,name,value);
    }
}while(parsed && !end_of_header);

/*Check that all headers exist*/
parsed = parsed && host_found;
return parsed ? OK : ERR;
}
```

## request.c (continued)

```
/*
 * Method that reads the content part of a http request header.
 * The method could be called several times to get the whole content.
 * Each call returns a chunk of data decided by parameter len.
 */
int request_read_content(Request *self, char *buf, int len)
{
    int res;

    assert(self);
    assert(buf);

    /* Read data from the socket */
    res = recv(self->_socket, buf, len, 0);

    /* Return 0 when there is no more data */
    if (res <= 0)
        return 0;
    else
        return res;
}

/*
 * Method that reads the header part of a http request header.
 * The header can be maximum HEADER_SIZE.
 */
Result request_read_header(Request *self)
{
    int len;
    int max;
    int line_end;
    int header_end;
    char header[HEADER_SIZE]; /* Input buffer */

    assert(self);

    max = line_end = header_end = 0;
    while (max < HEADER_SIZE - 1 && header_end != 1)
    {
        /*
         * Read one byte at the time from the socket and add it to the input buffer
         * By reading one byte at the time it is easy to check for end characters.
         */
        len = recv(self->_socket, &header[max], 1, 0);
        if (len == 0)
            return ERR; /* No data found */

        if (len < 0)
            return ERR; /* Socket closed */

        if (header[max] == '\n' && max > 1)
        {
            /* Here we have found a new line */
            if (header[max - 1] == '\r')
            {
                /* Previous character was carriage return */
                if (line_end)
                {
                    /*
                     * Here we have found a second \r\n directly after the previous
                     * so stop looking for header.
                     */
                    header_end = 1;
                }
                else
                {
                    /* First \r\n found */
                    line_end = 1;
                }
            }
        }
        else
        {
            if (header[max] != '\r')
            {
                /* Character is not \r so go back to start state. */
                line_end = 0;
            }
        }
        /* Move buffer index with number of bytes read. */
        max += len;
    }
}
```

## request.c (continued)

```
    }
    /* Header is found so terminate buffer*/
    header[max] = 0;

    /* Log the header if loglevel is 2*/
    logger_print(g_logger,2,"%s",header);

    /* If we found a proper header*/
    if (header_end)
    {
        /* Parse the header into its parts*/
        return parse_header(self,header);
    }
    else
    {
        /* Else return error*/
        return ERR;
    }
}
```

## response.h

```
/*
 * Declaration of the Response class.
 * This class is responsible for forming a correct
 * http response message and sending it.
 */

#ifndef _RESPONSE
#define _RESPONSE

#include "common.h"
#include "map.h"
#include "request.h"

/*
 * The Response data type.
 */
typedef struct Response
{
    SOCKET    _socket;
    char      _version[VERSION_SIZE];
    char      _phrase[PHRASE_SIZE];
    char      _status[STATUS_SIZE];
    Map       *_headers;
    int       _recursiveCalls;
} Response;

/* The Response methods*/
Response *response_create(SOCKET s);
void response_delete(Response *self);
Result response_send_file(Response *self, Request *req);
Result response_send_data(Response *self, Request *req, char *data, char *content_type);
Result response_send_not_found(Response *self);
Result response_send_not_implemented(Response *self);
Result response_send_internal_server_error(Response *self);
Result response_send_bad_request(Response *self);
Result response_send_moved_permanently(Response *self, char *location);
Result response_send_redirect(Response *self);
Result response_send_not_modified(Response *self);

#endif /* _RESPONSE */
```

## response.c

```
/*
 * Definition of the Response class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include "response.h"
#include "request.h"
#include "cgiprocess.h"
#include "cgihandler.h"
#include "logger.h"
#include "sessionhandler.h"
#include "directorybrowser.h"

/*
 * System global objects used by this class.
 */
/* Global path to the server executable*/
extern char g_module_path[256];
/* Global Sessionhandler object*/
extern Sessionhandler *g_sessionhandler;
/* Global variable showing if directory browsing is allowed*/
extern int g_conf_brows_default;
/* Global Cgihandler object*/
extern Cgihandler *g_cgihandler;

/* Define number of content types*/
#define MAX_CONTENT_TYPES 46

/* Declaration of content types that are
 * supported by the server.
 */
char *(content_types[MAX_CONTENT_TYPES][2]) = {
    {"htm", "text/html"},
    {"html", "text/html"},
    {"shtm", "text/html"},
    {"shtml", "text/html"},
    {"js", "application/x-javascript"},
    {"ico", "image/x-icon"},
    {"xml", "text/xml"},
    {"pdf", "application/pdf"},
    {"txt", "text/plain"},
    {"jpg", "image/jpeg"},
    {"jpeg", "image/jpeg"},
    {"png", "image/png"},
    {"bmp", "image/bmp"},
    {"svg", "image/svg+xml"},
    {"torrent", "application/x-bittorrent"},
    {"wav", "audio/x-wav"},
    {"mp3", "audio/x-mp3"},
    {"mid", "audio/mid"},
    {"m3u", "audio/x-mpegurl"},
    {"ram", "audio/x-pn-realaudio"},
    {"ra", "audio/x-pn-realaudio"},
    {"xls", "application/excel"},
    {"c", "text/plain"},
    {"h", "text/plain"},
    {"gif", "image/gif"},
    {"exe", "application/octet-stream"},
    {"zip", "application/zip"},
    {"rtf", "application/rtf"},
    {"css", "text/css"},
    {"wav", "audio/x-wav"},
    {"doc", "application/msword"},
    {"docx", "application/vnd.openxmlformats-officedocument.wordprocessingml.document"},
    {"xlsx", "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"},
    {"ppt", "application/ms-powerpoint"},
    {"bin", "application/octet-stream"},
    {"tgz", "application/x-tar-gz"},
    {"tar", "application/x-tar"},
    {"gz", "application/x-gunzip"},
    {"arj", "application/x-arj-compressed"},
    {"rar", "application/x-arj-compressed"},
    {"swf", "application/x-shockwave-flash"},
    {"mpg", "video/mpeg"},
    {"mpeg", "video/mpeg"},
    {"asf", "video/x-ms-asf"},
    {"avi", "video/x-msvideo"},
}
```



## response.c (continued)

```
{ "default", "text/plain" } };

/*
 * Month names
 */
static const char *month_names[] = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

/*
 * Constructor
 */
Response *response_create(SOCKET s)
{
    Response *new_obj;

    new_obj = malloc(sizeof(Response));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_socket = s;
        new_obj->_headers = map_create();
        new_obj->_recursiveCalls = 0;
    }
    return new_obj;
}

/*
 * Destructor
 */
void response_delete(Response *self)
{
    assert(self);
    assert(self->_headers);
    map_delete(self->_headers);
    free(self);
}

/*
 * Private method used to append a buffer.
 * The method returns the adress to to the byte after
 * the last character that was appended.
 * It is used to build a response header.
 */
static char *add_to_header(char *dest, char *src)
{
    char *d;
    char *s;

    assert(dest);
    assert(src);

    for (s = src, d = dest; *s != '\0'; s++, d++)
    {
        *d = *s;
    }
    return d;
}

/*
 * Private method used to build a response header
 * and send it with the response socket.
 */
static Result write_response_header(Response *self)
{
    char header[HEADER_SIZE];
    char *next_pos;
    Mapelement *e;

    assert(self);

    next_pos = header;
    /* HTTP Version */
    next_pos = add_to_header(next_pos, self->_version);
    next_pos = add_to_header(next_pos, " ");

    /* HTTP Status */
    next_pos = add_to_header(next_pos, self->_status);
    next_pos = add_to_header(next_pos, " ");
}
```

## response.c (continued)

```
/* HTTP Phrase*/
next_pos = add_to_header(next_pos,self->_phrase);

/* Add CRLF*/
next_pos = add_to_header(next_pos,"\r\n");

/* Add all header lines that response object includes*/
e = map_findfirst(self->_headers);
if (e != NULL)
{
    do
    {
        /* Add header line name*/
        next_pos = add_to_header(next_pos,e->key);
        next_pos = add_to_header(next_pos," ");

        /* Add header line value*/
        next_pos = add_to_header(next_pos,e->value);

        /* Add CRLF*/
        next_pos = add_to_header(next_pos,"\r\n");
        e = map_findnext(self->_headers);
    }while(e != NULL);

}
/* Add terminating CRLF*/
next_pos = add_to_header(next_pos,"\r\n");
/* NULL terminate string*/
*next_pos = '\0';

/* Log header to send if loglevel is 2*/
logger_print(g_logger,2,"%s",header);

/* Send the complete response header to the client*/
send(self->_socket,header,(int)(next_pos-header),0);

return OK;
}

/*
 * Method that sends a file as a http response content.
 * The method takes an opened file as parameter.
 */
static Result send_file_content(Response *self,FILE *f)
{
    char    file_buf[SEND_FILE_CONTENT_CHUNK_SIZE];
    size_t  size;

    assert(self);
    assert(f);

    /* Loop through the file in chunk-steps and send the chunks*/
    do
    {
        /* Read file data*/
        size = fread(file_buf,1,sizeof(file_buf),f);
        /* Send file data to the socket*/
        send(self->_socket,file_buf,(int)size,0);
    }while(!feof(f));

    return OK;
}

/*
 * Method that sends a buffer as a http response content io the open socket.
 */
static Result send_buffer_content(Response *self,char *data)
{
    char    buf[SEND_FILE_CONTENT_CHUNK_SIZE];
    char    *s;
    char    *send_start, *send_end;
    int     i;
    int     res;

    assert(self);
    assert(data);

    s = data;
    send_start = buf;
    while(*s != 0)
    {

```

## response.c (continued)

```
    send_end = buf;
    /*Fill send buffer with a chunk of data*/
    for(i = 0;i < SEND_FILE_CONTENT_CHUNK_SIZE && *s != 0;i++)
    {
        *send_end++ = *s++;
    }
    /* Send file data to the socket*/
    res = send(self->_socket,send_start,i,0);
}
return OK;
}

/*
 * Private method that returns a files extension.
 */
static char *get_file_extension(char *file,char *ext)
{
    char *p;
    char cnt;

    /* Pre NULL terminate result*/
    *ext = 0;

    assert(file);
    assert(ext);

    /* Loop through the file name*/
    for(p = file;*p != 0;p++)
    {
        /* Look for a dot*/
        if (*p == '.')
        {
            cnt = 0;
            /* If a dot is found assume that the comming text is the extension*/
            while (*(++p) && (cnt < FILE_EXTENSION - 1 ))
            {
                /* Copy the comming text to the result buffer*/
                cnt++;
                *(ext++) = *p;
            }
            /* NULL terminate string*/
            *ext = 0;
            break;
        }
    }
    /* Return the extension*/
    return ext;
}

/*
 * Private method that maps a file extension
 * to a http content-type by looking in a table.
 */
static char *get_content_type(char *ext,char *type)
{
    int index;

    assert(ext);
    assert(type);

    /* Loop through the content type table*/
    for (index = 0;index < MAX_CONTENT_TYPES;index++)
    {
        /* Search for the extension*/
        if (_stricmp(content_types[index][0],ext) == 0)
        {
            /* Extension found so return the corresponding content type*/
            strcpy(type,content_types[index][1]);
            return type;
        }
    }
    /*
     * No match in the table so return the default type
     * which is the last element in the table.
     */
    strcpy(type,content_types[MAX_CONTENT_TYPES - 1][1]);
    return type;
}
```

## response.c (continued)

```
/*
 * Private method that searches for a cookie in
 * a string and returns its value.
 */
static char *get_cookie_value(char *cookie_str, char *cookie_name, char *cookie_value)
{
    char *p;
    char *s;

    assert(cookie_str);
    assert(cookie_name);
    assert(cookie_value);

    /* Find first occurrence of the cookie name in the string*/
    p = strstr(cookie_str, cookie_name);
    s = cookie_value;

    /* If the name was found*/
    if (p != NULL)
    {
        /* Search forward to next '='*/
        while(*p != '=' && *p)
            p++;
        /* If we found a '=' before the end of the string*/
        if (*p == '=')
        {
            /* Collect the string as the value until a ';' is found*/
            p++;
            while(*p != ';' && *p)
                *s++ = *p++;
            /* NULL terminate the result string*/
            *s = 0;
            /* Return the result string*/
            return cookie_value;
        }
    }
    /* Not found so return NULL*/
    return NULL;
}

/*
 * Private method that determines if a path is a directory.
 */
static int is_directory(char *path)
{
    struct stat info;

    assert(path);
    /* Use the stat file system function*/
    if (!stat(path, &info))
    {
        /* Check if the directory flag is set*/
        if (info.st_mode & _S_IFDIR )
        {
            return 1; /* Is directory*/
        }
        return 0;
    }
    return 0;
}

/*
 * Private method that determines if a file exists on disk.
 */
static int file_exist(char *file_path)
{
    FILE *f;

    assert(file_path);

    f = NULL;
    /* Try to open the file*/
    f = fopen(file_path, "rb");
    if (!f)
        return 0; /* Could not open file*/
    else
    {
        /* Could open file so close it*/
        fclose(f);
        return 1;
    }
}
```

## response.c (continued)

```
}

/*
 * Private method that appends the right default file to a path.
 * It returns 1 if a default file was found and added else 0.
 * The default file is one of:
 * index.cgi (converts to index.exe)
 * index.php
 * index.pl
 * index.htm
 */
static int add_default_file(char *path)
{
    char file[FILE_SIZE];

    assert(path);

    strcpy(file,path);
    strcat(file,BINARY_DEFAULT_FILE);
    /* If index.exe exists index.cgi will be the default file*/
    if (file_exist(file))
    {
        strcat(path,CGI_DEFAULT_FILE);
        return 1;
    }

    strcpy(file,path);
    strcat(file,PHP_DEFAULT_FILE);
    /* If index.php exists it will be the default file*/
    if (file_exist(file))
    {
        strcat(path,PHP_DEFAULT_FILE);
        return 1;
    }

    strcpy(file,path);
    strcat(file,PERL_DEFAULT_FILE);
    /* If index.pl exists it will be the default file*/
    if (file_exist(file))
    {
        strcat(path,PERL_DEFAULT_FILE);
        return 1;
    }

    /* Else the default file will be index.htm*/
    strcpy(file,path);
    strcat(file,HTM_DEFAULT_FILE);
    if (file_exist(file))
    {
        strcat(path,HTM_DEFAULT_FILE);
        return 1;
    }

    return 0;
}

/*
 * Private method that tries to build a physial file path on disk from
 * the file in the request object.
 * If a file name is missing it appends a default file name.
 * If directory browsing is allowed it sends the directory content.
 */
static Result retrieve_physical_file_name_or_brows(Response *self,Request *req,char *physical)
{
    char file[FILE_SIZE];
    size_t last;
    char * host;
    Directorybrowser *dir_brows = NULL;
    int can_brows_dir = 0;
    int has_default_file;
    char *brows_data = 0;
    Result res;
    int is_root = 0;

    assert(self);
    assert(req);
    assert(physical);

    /* Add the server web path to the filename*/
    strcpy(file,g_module_path);
```

## response.c (continued)

```
    strcat(file, req->_file);

    /* Get the last character in the file name*/
    last = strlen(file) - 1;
    assert(last > 0);

    /* If the filename ends with a slash it means that a default file is requested*/
    if (file[last] == '\\\\' || file[last] == '/')
    {
        /* Create a directory browser object that can decide if to browse the directory*/
        dir_brows = directorybrowser_create(file, g_conf_brows_default, req->_file);
        can_brows_dir = directorybrowser_can_brows(dir_brows);

        has_default_file = add_default_file(file);

        if (can_brows_dir && !has_default_file)
        {
            /*Perform the directory browsing*/
            directorybrowser_get_brows_data(dir_brows, &brows_data);
            assert(brows_data);
            res = response_send_data(self, req, brows_data, "text/html");
        }
        directorybrowser_delete(dir_brows);
        if (OK == res)
        {
            return ENDED;
        }
    }
    /*
    * If no default file is requested go on and try to figure out
    * if the requested file is a physical file or a directory.
    */
    else if (is_directory(file))
    {
        /* If the requested file is a directory not ending with a slash (default file)
        * tell the browser that it should request the directory with a slash.
        * This is done by send the message MOVED PERMANENTLY to the browser
        * with the new file name including a slash.
        */
        /* Get the host name that is needed when building new URL*/
        host = map_getval(req->_headers, "Host:");
        if (host == NULL)
            return ERR;
        /* Start build a new URL*/
        strcpy(file, "http://");
        strcat(file, host);
        strcat(file, req->_file);
        /* Add a slash (default file)*/
        strcat(file, "/");
        /* Send the response*/
        response_send_moved_permanently(self, file);
        /* Return that the response is already taken care of*/
        return ENDED;
    }
    /* Return the physical file name*/
    strcpy(physical, file);
    return OK;
}

/*
* Private method that determines if a file name is a CGI script file.
*/
static int is_cgi(char *ext)
{
    assert(ext);
    return !_stricmp(ext, "cgi") || !_stricmp(ext, "php") || !_stricmp(ext, "pl");
}
```

## response.c (continued)

```
/*
 * Private method that replaces the cgi extension to executable.
 */
static void convert_to_executable(char *cgi_file)
{
    char *p;

    assert(cgi_file);

    /* Loops to the '.' in the file name*/
    for(p = cgi_file; *p != '.'; p++)
    {
        /* Empty loop just looking for '.'*/
    }
    p++;
    /* Replaces the extension with exe*/
    strcpy(p, "exe");
}

/*
 * Private method that builds the session id cookie string
 * from the session id in the request object.
 */
static char *build_sessionid_cookie_string(Request *req, char *create_cookie)
{
    char *host;

    assert(req);
    assert(create_cookie);

    /* Start building string*/
    strcpy(create_cookie, SESSION_ID_COOKIE_NAME);
    strcat(create_cookie, "=");
    /* Append the session id*/
    strcat(create_cookie, req->_session_id);
    /* Build the host part of the cookie by getting the host name*/
    host = map_getval(req->_headers, "Host:");
    /* Add the host part*/
    strcat(create_cookie, "; ");
    strcat(create_cookie, host);
    /* Return the new cookie string*/
    return create_cookie;
}

/*
 * Private method that creates a valid session id for the request
 */
static void create_session_id_from_cookie(Request *req)
{
    char *cookie;

    assert(req);

    /* Get the cookie string from the request*/
    cookie = map_getval(req->_headers, "Cookie:");
    /* If any cookie exists*/
    if (cookie != NULL)
    {
        /* Get the cookie value for cookie session id*/
        get_cookie_value(cookie, SESSION_ID_COOKIE_NAME, req->_session_id);
    }
    else
    {
        /* No cookie was sent from the client so just NULL terminate string*/
        req->_session_id[0] = 0;
    }

    /*
     * Now decide if the session id from the client still is
     * a valid id, else generate a new one with help of the session handler.
     */
    sessionhandler_get_valid_id(g_sessionhandler, req->_session_id);
}
```

## response.c (continued)

```
/*
 * Private method that parse the headers sent from a cgi script.
 */
static Result parse_cgi_response(Response *self, char *content, char **after_header, int *header_count)
{
    char *p;
    int cnt;
    int at_new_line;
    int collect;
    int token;
    char *t;
    char *s;
    int found_any_header = 0;

    assert(self);
    assert(content);

    /* Assume we are at the beginning of a script output line*/
    at_new_line = 1;
    /* Dont' collect header value yet*/
    collect = 0;
    /* Loop through the output in maximum CGI_HEADER_MAX number of characters*/
    for (p = content, cnt = 0; (cnt < CGI_HEADER_MAX) && (*p != 0); p++, cnt++)
    {
        /* If beginning of line*/
        if (at_new_line)
        {
            /* No token found yet*/
            token = 0;
            /* If header is Location:*/
            if (!strcmp("Location:", p, 9))
            {
                t = p;
                /* Signal header found*/
                token = 1;
                /* We are not in the state of beginning of a line*/
                at_new_line = 0;
                found_any_header = 1;
            }
            /* If header is Content-Type:*/
            else if (!strcmp("Content-Type:", p, 13))
            {
                t = p;
                /* Signal header found*/
                token = 2;
                /* We are not in the state of beginning of a line*/
                at_new_line = 0;
                found_any_header = 1;
            }
            /* If header is Set-Cookie:*/
            else if (!strcmp("Set-Cookie:", p, 11))
            {
                t = p;
                /* Signal header found*/
                token = 3;
                /* We are not in the state of beginning of a line*/
                at_new_line = 0;
                found_any_header = 1;
            }
            /* If we are at beginning of a line dont' collect header value*/
            collect = 0;
        }

        /* If state is collect*/
        if (collect)
        {
            /* Collect value by saving pointer to value*/
            s = p;
            /* Dont' collect any more*/
            collect = 0;
        }

        /* If header found and we have reached the end of the header name*/
        if (*p == ' ' && token > 0)
        {
            /* Set state collect*/
            collect = 1;
            /* NULL terminate header name string*/
            *p = 0;
        }
    }
}
```



## response.c (continued)

```
/* If new line is found*/
if (*p == '\r')
{
    /* If already at beginning of a line*/
    if (at_new_line)
    {
        /* We have found the end of the header*/
        /* Was any header found from the script*/
        if (found_any_header)
        {
            p++;
            p++;
            cnt += 2;
            /* Return the position coming after header*/
            *after_header = p;
            /* Return the size of the header*/
            *header_count = cnt;
            /* Return that we are done with success*/
            return OK;
        }
        else
        {
            /*
             * If no header was returned from the script return that
             * header size was 0 and return that the next position is
             * the start of the incoming content.
             */
            *header_count = 0;
            *after_header = content;
            /* Return that we are done with success*/
            return OK;
        }
    }
    /* New line was found but it was not the end of the header*/
    *p = 0;
    p++;
    cnt++;
    /* Set state to be beginning of line*/
    at_new_line = 1;
    /* Add the header name and value to the header map of the Response object*/
    map_add(self->headers,t,s);
}
}
/*
 * No data was returned from the script so return that
 * header size was 0 and return that the next position is
 * the start of the incoming content.
 */
*header_count = 0;
*after_header = content;
/* Return that we are done with success*/
return OK;
}

/*
 * Private method that takes the content from a cgi script
 * and sends a response of it.
 */
static Result send_script_response(Response *self, Request *req, char *content, int content_len)
{
    time_t      rawtime;
    struct tm    *timeinfo;
    char         http_time[80];
    char         size_buf[10];
    char         create_cookie[COOKIE_SIZE];
    int          has_content = 0;
    char         *parsed_content;
    int          header_count;
    Mapelement  *e;

    assert(self);
    assert(req);

    /* If there is any content*/
    if (content != NULL)
    {
        /* Parse the content for returned headers*/
        parse_cgi_response(self, content, &parsed_content, &header_count);
        /* Adjust the content size for headers found*/
        content_len -= header_count;
    }
}
```

## response.c (continued)

```
else
{
    /* Signal no parsed content*/
    parsed_content = NULL;
}

/* Handle script headers found specific*/
if ((e = map_find(self->_headers,"Location:")) != NULL)
{
    /* Handle the Location: header which means that the
     * cgi script wants to change location.
     */
    /* Check for absolut URI*/
    if (_strnicmp(e->value,"http://",6) == 0)
    {
        /* it is an absolute URI so make a 302 Redirect*/
        return response_send_redirect(self);
    }
    else
    {
        /* Else make an ordinary file request internally*/
        /* Take the file that was requested*/
        strcpy(req->_file,e->value);
        /* Remove the Location header from the script*/
        map_remove(self->_headers,"Location:");
        /* Remove the Content-Length header from the script*/
        map_remove(req->_headers,"Content-Length:");
        self->_recursiveCalls++;
        /* Check number of recursive location changes*/
        if (self->_recursiveCalls < RECURSIVE_CALLS_ALLOWED)
        {
            /* Make a file response*/
            return response_send_file(self,req);
        }
        else
        {
            /* To many recursive calls so return bad request*/
            return response_send_bad_request(self);
        }
    }
}

/* Set the http response status line to 200 OK*/
strcpy(self->_version,"HTTP/1.1");
strcpy(self->_status,"200");
strcpy(self->_phrase,"OK");

/* Set the Date header by retrieving the system time*/
time ( &rawtime );
timeinfo = localtime( &rawtime );
strftime(http_time,sizeof(http_time),"%a, %d %b %Y %H:%M:%S GMT",timeinfo);
map_add(self->_headers,"Date:",http_time);
/* Set server header*/
map_add(self->_headers,"Server:", "minaliC 1.0");
/* Set Connection header (always close)*/
map_add(self->_headers,"Connection:", "Close");
/* Set the session cookie*/
map_add(self->_headers,"Set-Cookie:",build_sessionid_cookie_string(req,create_cookie));

/* If the response includes any content*/
if (content_len > 0)
{
    /* And if we got any content from the script*/
    if (parsed_content != NULL)
    {
        /* Set the content-length header*/
        _ltoa(content_len,size_buf,10);
        map_add(self->_headers,"Content-Length:",size_buf);
        has_content = 1;
    }
    /*
     * Set the content-type header.
     * If the script returned a content-type header use that one,
     * else assume that the content from a script is text/html.
     */
    if (map_find(self->_headers,"Content-Type:") == NULL)
    {
        map_add(self->_headers,"Content-Type:", "text/html");
    }
}
```

## response.c (continued)

```
/* If we could send the header with success*/
if (write_response_header(self) == OK)
{
    /* If there was any content to send...*/
    if (has_content)
    { /* ...send that content*/
        if (send(self->_socket,parsed_content,content_len,0) == content_len)
        {
            /* Return sent OK*/
            return OK;
        }
    }
    else
    {
        /* Return sent OK (no content sent, just a header)*/
        return OK;
    }
}
/* If we got here we could not send the response*/
return ERR;
}

/*
 * Private method that starts a cgi script response.
 */
static Result start_cgi_response(Response *self,Request *req,char *web_file,char *ext)
{
    Cgiprocess *cgi;
    HANDLE      cgi_out = NULL;
    char        *cgi_buffer = NULL;
    long        index = 0;
    long        cnt = 0;
    int         pages = 4;
    int         len;
    Result      res;
    int         read_result;
    char        tmp[256];
    int         timed_out;

    assert(self);
    assert(req);
    assert(web_file);
    assert(ext);

    /* Check so the extension of the script file not is php or pl(perl)*/
    if (!_stricmp(ext,"php") && !_stricmp(ext,"pl"))
    {
        /* Convert *.cgi to correct extension*/
        convert_to_executable(web_file);
    }

    /* Physical script file must exist*/
    if (file_exist(web_file))
    {
        /* If the extension of the script file is php*/
        if (!_stricmp(ext,"php"))
        {
            /*
             * Build a path to the php cli binary and
             * let that binary be the cgi script to start.
             */
            strcpy(tmp,"php.exe ");
            strcat(tmp," \\");
            strcat(tmp,web_file);
            strcat(tmp," \\");
            strcpy(web_file,tmp);
        }
        else if (!_stricmp(ext,"pl"))
        {
            /*
             * Build a path to the perl cli binary and
             * let that binary be the cgi script to start.
             */
            strcpy(tmp,"perl.exe ");
            strcat(tmp," \\");
            strcat(tmp,web_file);
            strcat(tmp," \\");
            strcpy(web_file,tmp);
        }
    }
}
```

## response.c (continued)

```
/* Create an instance of a Cgiprocess*/
cgih = cgiprocess_create(req,web_file);
assert(cgih);
/*
 * Let the Cgiprocess start the script and
 * handle all communication with it. The call will
 * return a stdout handle from the script.
 */
cgi_out = cgiprocess_invoke(cgih);
assert(cgih);
/* If we got a valid stdout handle*/
if (cgi_out != NULL)
{
    /* Register activity for the cgi process*/
    cgihandler_set_active(g_cgihandler,cgih->_pid);
    /* Allocate initial size of receive buffer*/
    cgi_buffer = malloc(CGI_BUFFER_SIZE * pages);
    /* Loop until we have read all content from the script*/
    do
    {
        /* Read from scripts stdout into the buffer*/
        read_result = ReadFile(cgi_out,&cgi_buffer[index],CGI_BUFFER_SIZE,&len,NULL);
        if (read_result)
        {
            /* Register activity for the cgi process*/
            cgihandler_set_active(g_cgihandler,cgih->_pid);
            /* Increase index in buffer*/
            index += len;
            /* count the number of bytes read*/
            cnt += len;

            /* If number of bytes read is on its way to reach the buffer limit*/
            if (cnt >= CGI_BUFFER_SIZE * (pages - 1))
            {
                /* Count up the number of allocated pages*/
                pages++;
                /* Allocate more memory for the receive buffer*/
                cgi_buffer = realloc(cgi_buffer,CGI_BUFFER_SIZE * pages);
            }
        }
        /* Loop until no more to read*/
    }while(read_result && len);

    /* NULL terminate the receive buffer*/
    cgi_buffer[cnt] = 0;

    /* Unregister the cgi process and read timeout result*/
    timed_out = cgihandler_remove(g_cgihandler,cgih->_pid);
    if (!timed_out)
    {
        /* Send the content from the script in a response to the client*/
        res = send_script_response(self,req,cgi_buffer,cnt);
    }
    else
    {
        /* Send cgi timeout error as not found*/
        res = response_send_not_found(self);
    }
    assert(cgi_buffer);
    /* Free memory for the receive buffer*/
    free(cgi_buffer);
    /* Delete the Cgi handler instance*/
    cgiprocess_delete(cgih);
    /* Return the result from the send operation*/
    return res;
}
else
{
    return ERR;
}
}
/* If the script file doesn't exist send NOT_FOUND*/
return NOT_FOUND;
}

/*
 * Private function that convert month to the month number. Return -1 on error.
 */
static int monnum(const char *s)
{
    size_t i;
```

## response.c (continued)

```
    for (i = 0; i < sizeof(month_names) / sizeof(month_names[0]); i++)
        if (!strcmp(s, month_names[i]))
            return ((int) i);

    return (-1);
}

/*
 * Private function that return the corresponding time_t value for a webdate.
 */
static time_t webdate_to_time(const char *s)
{
    time_t    current_time;
    struct tm  tm, *tmp;
    char      mon[32];
    int       sec, min, hour, mday, month, year;

    memset(&tm, 0, sizeof(tm));
    sec = min = hour = mday = month = year = 0;

    if (((sscanf(s, "%d/%3s/%d %d:%d:%d",
                &mday, mon, &year, &hour, &min, &sec) == 6) ||
        (sscanf(s, "%d %3s %d %d:%d:%d",
                &mday, mon, &year, &hour, &min, &sec) == 6) ||
        (sscanf(s, "%*3s, %d %3s %d %d:%d:%d",
                &mday, mon, &year, &hour, &min, &sec) == 6) ||
        (sscanf(s, "%d-%3s-%d %d:%d:%d",
                &mday, mon, &year, &hour, &min, &sec) == 6)) &&
        (month = monnum(mon)) != -1)
    {
        tm.tm_mday = mday;
        tm.tm_mon  = month;
        tm.tm_year = year;
        tm.tm_hour = hour;
        tm.tm_min  = min;
        tm.tm_sec  = sec;
    }

    if (tm.tm_year > 1900)
        tm.tm_year -= 1900;
    else if (tm.tm_year < 70)
        tm.tm_year += 100;

    /* Set Daylight Saving Time field */
    current_time = time(NULL);
    tmp = localtime(&current_time);
    tm.tm_isdst = tmp->tm_isdst;

    return (mktime(&tm));
}

/*
 * Main function that handle all requests of type GET and POST.
 * The function looks to see if it is a cgi request or
 * a static file request.
 */
Result response_send_file(Response *self, Request *req)
{
    FILE      *f;
    struct stat info;
    time_t    rawtime;
    struct tm  *timeinfo;
    struct tm  file_timeinfo;
    char      http_time[HTTP_TIME_SIZE];
    char      size_buf[20];
    Result     res;
    char      web_file[2*FILE_SIZE];
    char      ext[FILE_EXTENSION];
    char      content_type[CONTENT_TYPE_SIZE];
    char      create_cookie[COOKIE_SIZE];
    char      *if_modified_since;

    assert(self);
    assert(req);
}
```

## response.c (continued)

```
/*
 * Translate the url request to a physical file
 * If the function returns ENDED the request was redirected.
 */
if (retrieve_physical_file_name_or_brows(self, req, web_file) == ENDED)
    return OK; /* User was redirected*/

/* Get the file extension of the requested file or cgi module*/
get_file_extension(web_file, ext);

/* Create or reuse the session id in the request*/
create_session_id_from_cookie(req);

/* Figure out if the requested file will generate a cgi call*/
if (is_cgi(ext))
{
    /* The file is a cgi module to invoke the cgi module requested
     * instead of sending a static file response.
     */
    return start_cgi_response(self, req, web_file, ext);
}

/* Else perform ordinary static file response*/
/* Try to open the requested file*/
f = fopen(web_file, "rb");
if (!f)
    return NOT_FOUND; /* File doesn't exist*/

/* Get timestamp and size of the file*/
stat(web_file, &info);
_ltoa(info.st_size, size_buf, 10);
localtime_s(&file_timeinfo, &(info.st_atime));

/* Check if we should use the If-Modified-since header*/
if_modified_since = map_getval(req->_headers, "If-Modified-Since:");
if (if_modified_since)
{
    /* Check if file is newer then if_modified_since*/
    if (info.st_atime <= webdate_to_time(if_modified_since))
    {
        /* Close file and send not_modified*/
        fclose(f);
        return response_send_not_modified(self);
    }
}

/* The file exist so build a response with the file*/
/* Create HTTP status line*/
strcpy(self->_version, "HTTP/1.1");
strcpy(self->_status, "200");
strcpy(self->_phrase, "OK");

/* Create a Date header*/
time(&rawtime);
timeinfo = localtime(&rawtime);
strftime(http_time, sizeof(http_time), "%a, %d %b %Y %H:%M:%S GMT", timeinfo);
map_add(self->_headers, "Date:", http_time);

/* Create a content-type header*/
map_add(self->_headers, "Content-Length:", size_buf);
get_content_type(ext, content_type);
map_add(self->_headers, "Content-Type:", content_type);

/* Create a last-modified header with the files date and time*/
strftime(http_time, sizeof(http_time), "%a, %d %b %Y %H:%M:%S GMT", &file_timeinfo);
map_add(self->_headers, "Last-Modified:", http_time);

/* Create the server header*/
map_add(self->_headers, "Server:", "minaliC 1.0");

/* Create the connection header. Always use the close type*/
map_add(self->_headers, "Connection:", "Close");

/* Create a set-cookie header for the session id*/
map_add(self->_headers, "Set-Cookie:", build_sessionid_cookie_string(req, create_cookie));

/* Send the status line and the headers*/
if (write_response_header(self) == OK)
{
    /* Then send the requested file content*/
    res = send_file_content(self, f);
    fclose(f);
}
```

## response.c (continued)

```
    return res;
}
fclose(f);

/* If we get here no successfull response could be made*/
return ERR;
}

/*
 * Private method that sends a simple response without any content.
 */
static Result response_send_msg(Response *self, char *status, char *phrase)
{
    time_t      rawtime;
    struct tm    *timeinfo;
    char        http_time[80];

    assert(self);
    /* Set the status line*/
    strcpy(self->_version, "HTTP/1.1");
    strcpy(self->_status, status);
    strcpy(self->_phrase, phrase);
    /* Set the date header*/
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(http_time, sizeof(http_time), "%a, %d %b %Y %H:%M:%S GMT", timeinfo);
    map_add(self->_headers, "Date:", http_time);
    /* Send the response header and return*/
    return write_response_header(self);
}

/*
 * Private method that sends a response with content from a buffer.
 * The content buffer should be null terminated
 */
Result response_send_data(Response *self, Request *req, char *data, char *content_type)
{
    time_t      rawtime;
    struct tm    *timeinfo;
    char        http_time[HTTP_TIME_SIZE];
    Result      res;
    char        size_buf[20];
    char        create_cookie[COOKIE_SIZE];

    assert(self);
    assert(req);
    assert(data);

    /* Create or reuse the session id in the request*/
    create_session_id_from_cookie(req);

    /* Create HTTP status line*/
    strcpy(self->_version, "HTTP/1.1");
    strcpy(self->_status, "200");
    strcpy(self->_phrase, "OK");

    /* Create a Date header*/
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(http_time, sizeof(http_time), "%a, %d %b %Y %H:%M:%S GMT", timeinfo);
    map_add(self->_headers, "Date:", http_time);

    /* Create a content-type header*/
    _ltoa((unsigned long)strlen(data), size_buf, 10);
    map_add(self->_headers, "Content-Length:", size_buf);
    map_add(self->_headers, "Content-Type:", content_type);

    /* Create the server header*/
    map_add(self->_headers, "Server:", "minaliC 1.0");

    /* Create the connection header. Always use the close type*/
    map_add(self->_headers, "Connection:", "Close");

    /* Create a set-cookie header for the session id*/
    map_add(self->_headers, "Set-Cookie:", build_sessionid_cookie_string(req, create_cookie));
}
```

## response.c (continued)

```
/* Send the status line and the headers*/
if (write_response_header(self) == OK)
{
    /* Then send the requested file content*/
    res = send_buffer_content(self,data);
    return res;
}

/* If we get here no successfull response could be made*/
return ERR;
}

/*
 * Method that sends a simple response of type 404.
 */
Result response_send_not_found(Response *self)
{
    return response_send_msg(self,"404","Not Found");
}

/*
 * Method that sends a simple response of type 501.
 */
Result response_send_not_implemented(Response *self)
{
    return response_send_msg(self,"501","Not Implemented");
}

/*
 * Method that sends a simple response of type 500.
 */
Result response_send_internal_server_error(Response *self)
{
    return response_send_msg(self,"500","Internal server error");
}

/*
 * Method that sends a simple response of type 400.
 */
Result response_send_bad_request(Response *self)
{
    return response_send_msg(self,"400","Bad Request");
}

/*
 * Method that sends a simple response of type 300.
 */
Result response_send_moved_permanently(Response *self,char *location)
{
    /* Add to the response the new location*/
    map_add(self->headers,"Location:",location);
    return response_send_msg(self,"300","Moved Permanently");
}

/*
 * Method that sends a simple response of type 302.
 */
Result response_send_redirect(Response *self)
{
    return response_send_msg(self,"302","Redirect");
}

/*
 * Method that sends a simple response of type 304.
 */
Result response_send_not_modified(Response *self)
{
    return response_send_msg(self,"304","Not Modified");
}
```



## cgiprocess.h

```
/*
 * Declaration of the Cgiprocess class.
 * This class handles all tricky things of
 * creating a cgi script process and communicating with is.
 */

#ifndef _CGIPROCESS
#define _CGIPROCESS

#include "request.h"
#include "response.h"

/*
 * The Cgiprocess data type.
 */
typedef struct Cgiprocess
{
    /* Reference to a Request object*/
    Request *_request;
    /* The name of the cgi module to run*/
    char *_module;
    /* Cgi module standard input to read from*/
    HANDLE _child_stdin_read;
    /* Cgi module standard input to write to*/
    HANDLE _child_stdin_write;
    /* Cgi module standard output to read from*/
    HANDLE _child_stdout_read;
    /* Cgi module standard output to write to*/
    HANDLE _child_stdout_write;
    /* Process id of the cgi process*/
    unsigned long _pid;
} Cgiprocess;

/* The Cgiprocess methods*/
Cgiprocess *cgiprocess_create(Request *req, char *module);
void cgiprocess_delete(Cgiprocess *self);
HANDLE cgiprocess_invoke(Cgiprocess *self);

#endif /* _CGIPROCESS*/
```

## cgiprocess.c

```
/*
 * Definition of the Cgiprocess class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include "cgiprocess.h"
#include "request.h"
#include "map.h"
#include "logger.h"

/* Size of chunk buffer to read http content data in*/
#define CONTENT_CHUNK_BUFSIZE 4096
/* Size of buffer to create environment variables for cgi*/
#define ENVIRONMENT_AREA_SIZE 2048

/* Declaration of global server module path*/
extern char g_module_path[256];

/*
 * Constructor
 */
Cgiprocess *cgiprocess_create(Request *req, char *module)
{
    Cgiprocess *new_obj;
    assert(req);
    assert(module);
    new_obj = malloc(sizeof(Cgiprocess));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->request = req;
        new_obj->_module = malloc(strlen(module)+1);
        strcpy(new_obj->_module, module);
        new_obj->_child_stdin_read = NULL;
        new_obj->_child_stdin_write = NULL;
        new_obj->_child_stdout_read = NULL;
        new_obj->_child_stdout_write = NULL;
    }
    return new_obj;
}

/*
 * Destructor
 */
void cgiprocess_delete(Cgiprocess *self)
{
    assert(self);
    if (self)
    {
        CloseHandle(self->_child_stdin_read);
        CloseHandle(self->_child_stdout_read);
        free(self->_module);
        free(self);
    }
}

/*
 * Function that creates the process of the cgi module
 * and send the http content to its standard input.
 * The function returns a standard output handle to read
 * the module output content from.
 */
HANDLE cgiprocess_invoke(Cgiprocess *self)
{
    SECURITY_ATTRIBUTES sa_attr;
    PROCESS_INFORMATION proc_info;
    STARTUPINFOA start_info;
    BOOL success = FALSE;
    DWORD written;
    char chBuf[CONTENT_CHUNK_BUFSIZE];
    DWORD flags = 0;
    HANDLE input_file = NULL;
    int res;
    char new_env[ENVIRONMENT_AREA_SIZE];
    char *current_variable;
    char system_root[100];
    char system_root_value[100];
    char comspec[100];
    char comspec_value[100];
```

## cgiprocess.c (continued)

```
Mapelement          *e;
int                  content_length = 0;
int                  read_content = 0;

sa_attr.nLength = sizeof(SEcurity_ATTRIBUTES);
sa_attr.bInheritHandle = TRUE;
sa_attr.lpSecurityDescriptor = NULL;

/* Child process's STDOUT*/
if (!CreatePipe(&self->_child_stdout_read,&self->_child_stdout_write,&sa_attr,0))
    logger_print(g_logger,1,"StdoutRd CreatePipe");

/*Ensure the read handle to the pipe for STDOUT is not inherited*/
if (!SetHandleInformation(self->_child_stdout_read,HANDLE_FLAG_INHERIT,0))
    logger_print(g_logger,1,"Stdout SetHandleInformation");

/* Child process's STDIN*/
if (!CreatePipe(&self->_child_stdin_read,&self->_child_stdin_write,&sa_attr,0))
    logger_print(g_logger,1,"Stdin CreatePipe");

/* Ensure the write handle to the pipe for STDIN is not inherited*/
if (!SetHandleInformation(self->_child_stdin_write,HANDLE_FLAG_INHERIT,0))
    logger_print(g_logger,1,"Stdin SetHandleInformation");

/* Set up members of the PROCESS_INFORMATION structure*/
ZeroMemory(&proc_info,sizeof(PROCESS_INFORMATION));

/*Copy environment strings into an environment block*/
/* First create the system root variable that all processes
 * must have to work.
 */
GetEnvironmentVariableA("systemRoot",system_root,99);
strcpy(system_root_value,"systemRoot=");
strcat(system_root_value,system_root);

GetEnvironmentVariableA("COMSPEC",comspec,99);
strcpy(comspec_value,"comspec=");
strcat(comspec_value,comspec);

/* Add environment variable REQUEST_METHOD*/
current_variable = new_env;
strcpy(current_variable,"REQUEST_METHOD=");
strcat(current_variable,self->_request->_method);
current_variable += strlen(current_variable) + 1;

/* Add environment variable QUERY_STRING*/
strcpy(current_variable,"QUERY_STRING=");
strcat(current_variable,self->_request->_query);
current_variable += strlen(current_variable) + 1;

/* Add environment variable SERVER_SOFTWARE*/
strcpy(current_variable,"SERVER_SOFTWARE=minaliC 1.0");
current_variable += strlen(current_variable) + 1;

/* Add environment variable SERVER_NAME*/
strcpy(current_variable,"SERVER_NAME=minaliC 1.0");
current_variable += strlen(current_variable) + 1;

/* Add environment variable SCRIPT_NAME*/
strcpy(current_variable,"SCRIPT_NAME=");
strcat(current_variable,self->_request->_file);
current_variable += strlen(current_variable) + 1;

/* Add environment variable SCRIPT_FILENAME for PHP*/
strcpy(current_variable,"SCRIPT_FILENAME=");
strcat(current_variable,self->_request->_file);
current_variable += strlen(current_variable) + 1;

/* Add environment variable GATEWAY_INTERFACE*/
strcpy(current_variable,"GATEWAY_INTERFACE=CGI/1.1");
current_variable += strlen(current_variable) + 1;

/* Add environment variable SERVER_PROTOCOL*/
strcpy(current_variable,"SERVER_PROTOCOL=HTTP/1.1");
current_variable += strlen(current_variable) + 1;

/* Add environment variable REMOTE_ADDR*/
strcpy(current_variable,"REMOTE_ADDR=");
strcat(current_variable,inet_ntoa(self->_request->_addr.sin_addr));
current_variable += strlen(current_variable) + 1;
```

## cgiprocess.c (continued)

```
/* Add environment variable SESSION_ID*/
strcpy(current_variable,"SESSION_ID=");
strcat(current_variable,self->_request->_session_id);
current_variable += strlen(current_variable) + 1;

/* Add environment variable WWWROOT*/
strcpy(current_variable,"WWWROOT=");
strcat(current_variable,g_module_path);
current_variable += strlen(current_variable) + 1;

/* Loop through the request headers and add some selected headers
 * to the cgi modules' environment variables.
 */
e = map_findfirst(self->_request->_headers);
if (e != NULL)
{
    do
    {
        if (_stricmp(e->key,"Accept:") == 0)
        {
            /* Add environment variable HTTP_ACCEPT*/
            strcpy(current_variable,"HTTP_ACCEPT=");
            strcat(current_variable,e->value);
            current_variable += strlen(current_variable) + 1;
        }
        else if (_stricmp(e->key,"Content-Type:") == 0)
        {
            /* Add environment variable CONTENT_TYPE*/
            strcpy(current_variable,"CONTENT_TYPE=");
            strcat(current_variable,e->value);
            current_variable += strlen(current_variable) + 1;
        }
        else if (_stricmp(e->key,"Content-Length:") == 0)
        {
            /* Add environment variable CONTENT_LENGTH*/
            strcpy(current_variable,"CONTENT_LENGTH=");
            strcat(current_variable,e->value);
            /* Save the content length to compare with later*/
            content_length = atoi(e->value);
            current_variable += strlen(current_variable) + 1;
        }
        else if (_stricmp(e->key,"Cookie:") == 0)
        {
            /* Add environment variable HTTP_COOKIE*/
            strcpy(current_variable,"HTTP_COOKIE=");
            strcat(current_variable,e->value);
            current_variable += strlen(current_variable) + 1;
        }
        e = map_findnext(self->_request->_headers);
    }while(e != NULL);
}

/* Add environment variable COMSPEC*/
strcpy(current_variable,comspec_value);
/* Add environment variable SystemRoot*/
strcpy(current_variable,system_root_value);
/* Terminate the block with a NULL byte*/
current_variable += strlen(current_variable) + 1;
*current_variable = 0;

/* Set up members of the STARTUPINFO structure.
 * This structure specifies the STDIN and STDOUT handles for redirection.
 */
ZeroMemory(&start_info,sizeof(STARTUPINFO));
start_info.cb = sizeof(STARTUPINFO);
start_info.hStdError = self->_child_stdout_write;
start_info.hStdOutput = self->_child_stdout_write;
start_info.hStdInput = self->_child_stdin_read;
start_info.dwFlags |= STARTF_USESTDHANDLES;

success = CreateProcessA(NULL,
    self->_module, /*Command line*/
    NULL, /*Process security attributes*/
    NULL, /*Primary thread security attributes*/
    TRUE, /*Handles are inherited*/
    flags, /*Creation flags*/
    (LPVOID) new_env, /*New environment variables from parent*/
    NULL, /*Use parent's current directory*/
    &start_info, /*STARTUPINFO pointer*/
    &proc_info); /*Receives PROCESS_INFORMATION*/
```

## cgiprocess.c (continued)

```
if (!success)
{
    /* If an error occurs, send it to the log*/
    logger_print(g_logger,1,"Error creating CGI process %s",self->_module);
    /* Return a null handle*/
    return NULL;
}
else
{
    /* If process created successfully close the handles*/
    CloseHandle(proc_info.hProcess);
    CloseHandle(proc_info.hThread);
    /* Save the process id*/
    self->_pid = proc_info.dwProcessId;
}

/* If there is any content to read*/
if (content_length > 0)
{
    /* Read content from socket and send it to process stdin*/
    while (res = request_read_content(self->_request,chBuf,CONTENT_CHUNK_BUFSIZE))
    {
        /* Count the content read*/
        read_content += res;
        success = WriteFile(self->_child_stdin_write,chBuf,res,&written,NULL);
        /* If we have reached the expected content break*/
        if (read_content >= content_length)
            break;
    }
}

/* Close the pipe handles*/
if (!CloseHandle(self->_child_stdin_write))
    logger_print(g_logger,1,"StdInWr CloseHandle");

if (!CloseHandle(self->_child_stdout_write))
    logger_print(g_logger,1,"StdOutWr CloseHandle");

/* Return to the caller the cgi process' standard output handle*/
return self->_child_stdout_read;
}
```

## cgihandler.h

```
/*
 * Declaration of the Cgihandler class.
 * This class handles the timeout of cgi processes.
 */

#ifndef _CGIHANDLER
#define _CGIHANDLER

#include "request.h"
#include "response.h"

/*
 * The Cgihandler data type.
 */
typedef struct Cgihandler
{
    int _timeout;
    Map *_processes;
    /* Cgihandler is used by several threads and need a critical section*/
    CRITICAL_SECTION _critical_section;
} Cgihandler;

/* The Cgihandler methods*/
Cgihandler *cgihandler_create();
void cgihandler_delete(Cgihandler *self);
void cgihandler_set_active(Cgihandler *self, unsigned long pid);
int cgihandler_remove(Cgihandler *self, unsigned long pid);
void cgihandler_delete_old(Cgihandler *self);

#endif /*_CGIHANDLER*/
```

## cgihandler.c

```
/*
 * Definition of the Cgihandler class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include "cgihandler.h"
#include "map.h"
#include "logger.h"

/*
 * Constructor
 */
Cgihandler *cgihandler_create(int seconds)
{
    Cgihandler *new_obj;
    new_obj = malloc(sizeof(Cgihandler));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_timeout = seconds;
        new_obj->_processes = map_create();
        /* Initialize the critical section object*/
        InitializeCriticalSection(&new_obj->_critical_section);
    }
    return new_obj;
}

/*
 * Destructor
 */
void cgihandler_delete(Cgihandler *self)
{
    assert(self);
    assert(self->_processes);
    map_delete(self->_processes);
    DeleteCriticalSection(&self->_critical_section);
    free(self);
}

/*
 * Method used to set or update that a cgi process is alive.
 * The method can be called repeatedly to set that the cgi
 * process is active.
 */
void cgihandler_set_active(Cgihandler *self, unsigned long pid)
{
    time_t now;
    time_t then;
    char then_buf[20];
    char pid_buf[20];
    char *timeout;

    assert(self);

    /*
     * This is a common function called by several server threads so
     * it needs to be protected by a system critical section.
     */
    EnterCriticalSection(&self->_critical_section);

    /* Get current time*/
    time ( &now );
    then = now + self->_timeout;
    _ltoa((long)then, then_buf, 10);
    _ltoa(pid, pid_buf, 10);
    /* Read current timeout value for the process from the process map*/
    timeout = map_getval(self->_processes, pid_buf);
    /* If timeout value exist we must check it for 0*/
    if (timeout != NULL)
    {
        /* If the process already has be terminated don't update timeout value*/
        if (strcmp(timeout, "0") == 0)
        {
            LeaveCriticalSection(&self->_critical_section);
            return;
        }
    }
    /* Update/create process id with current time*/
}
```

## cgihandler.c (continued)

```
map_set(self->_processes,pid_buf,then_buf);

LeaveCriticalSection(&self->_critical_section);
}

/*
 * Method used to remove cgi process from process map.
 * The method return whether the process was timed out or not.
 */
int cgihandler_remove(Cgihandler *self, unsigned long pid)
{
    char *timeout;
    char pid_buf[20];
    int was_timedout;

    /*
     * This is a common function called by several server threads so
     * it needs to be protected by a system critical section.
     */
    EnterCriticalSection(&self->_critical_section);

    _ltoa(pid,pid_buf,10);
    timeout = map_getval(self->_processes,pid_buf);
    assert(timeout);
    /* If the process was timed out the process' timeout value
     * is set to 0 by the scheduler and here we check that value.
     */
    was_timedout = (timeout[0] == '0');
    /* Remove from the map*/
    map_remove(self->_processes,pid_buf);

    LeaveCriticalSection(&self->_critical_section);

    return was_timedout;
}

/*
 * Method that terminates all cgi processes that has timed out.
 * This method is called repeatedly by the server main function.
 */
void cgihandler_delete_old(Cgihandler *self)
{
    Mapelement *e = NULL;
    time_t now = 0;
    time_t then = 0;
    unsigned long pid;
    unsigned long exitcode;
    HANDLE hprocess;

    /*
     * This is a common function called by several threads so
     * it needs to be protected by a system critical section.
     */
    EnterCriticalSection(&self->_critical_section);
    assert(self);
    /* Get current time*/
    time ( &now );
    assert(now);
    /* Get first process id from map*/
    e = map_findfirst(self->_processes);
    if (e != NULL)
    {
        /* Loop through all processes to se if they have timed out*/
        do
        {
            /* Get current process timeout limit*/
            then = atol(e->value);
            /* Only check not terminated processes*/
            if (then > 0)
            {
                /* Compare the process timeout limit to current time*/
                if (then - now < 0)
                {
                    /* Set timeout value to 0, which means that
                     * the process has been terminated.
                     */
                    map_set_element_value(e,"0");
                    pid = atol(e->key);

                    /* Now terminate the cgi process*/

```



## cgihandler.c (continued)

```
        if (hprocess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,pid))
        {
            GetExitCodeProcess(hprocess, &exitcode);
            if (exitcode == STILL_ACTIVE)
                TerminateProcess(hprocess, 1);
            CloseHandle(hprocess);
        }
    }
    /* Get next session id*/
    e = map_findnext(self->_processes);
}while(e != NULL);
}
/* End of critical section*/
LeaveCriticalSection(&self->_critical_section);
}
```

## sessionhandler.h

```
/*
 * Declaration of the Sessionhandler class.
 * This class takes care of holding all session id's.
 * It knows when a session has timed out and how
 * to generate new session id's.
 */

#ifndef _SESSIONHANDLER
#define _SESSIONHANDLER

#include <windows.h>
#include "common.h"
#include "map.h"

/*
 * The Sessionhandler data type.
 */
typedef struct Sessionhandler
{
    int _timeout;
    Map *_sessions;
    /* Sessionhandler is used by several threads and need a critical section*/
    CRITICAL_SECTION _critical_section;
} Sessionhandler;

/* The Sessionhandler methods*/
Sessionhandler *sessionhandler_create();
void sessionhandler_delete(Sessionhandler *self);
int sessionhandler_get_valid_id(Sessionhandler *self, char *session_id);
void sessionhandler_delete_old(Sessionhandler *self);

#endif /* _SESSIONHANDLER*/
```

## sessionhandler.c

```
/*
 * Definition of the Sessionhandler class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "sessionhandler.h"
#include "common.h"
#include "map.h"
#include "logger.h"

/*
 * Constructor
 */
Sessionhandler *sessionhandler_create(int seconds)
{
    Sessionhandler *new_obj;
    new_obj = malloc(sizeof(Sessionhandler));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_timeout = seconds;
        new_obj->_sessions = map_create();
        /* Initialize the critical section object*/
        InitializeCriticalSection(&new_obj->_critical_section);
    }
    return new_obj;
}

/*
 * Destructor
 */
void sessionhandler_delete(Sessionhandler *self)
{
    assert(self);
    assert(self->_sessions);
    map_delete(self->_sessions);
    DeleteCriticalSection(&self->_critical_section);
    free(self);
}

/*
 * Private method that generates a new session id.
 */
static char *generate_id(char *session_id )
{
    int i;
    /* Buffer to pick random characters from*/
    char *character = "QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm0123456789";

    assert(session_id);
    /* Initialize the random generator with the time*/
    srand( (unsigned) clock() + (int)time( NULL ) );

    /* Loop through the buffer and get a random value for each character*/
    for ( i = 0; i < SESSION_ID_SIZE-1; i++ )
        /* Map the random value to a character in the buffer*/
        session_id[i] = character[rand()%62];

    /* NULL terminate the result*/
    session_id[i] = '\0';
    return session_id;
}

/*
 * Private method that tries to generate a unique session id
 * by calling generate_id() one or several times.
 */
static void get_new_session_id(Sessionhandler *self, char *session_id)
{
    time_t now;
    int cnt = 0;
    char now_buf[20];

    assert(self);
    assert(session_id);
    /* Start the try loop of generating unique id*/
    do
    {
        /* Count the tries*/

```

## sessionhandler.c (continued)

```
        cnt++;
        /* Get the current time of the generation*/
        time(&now);
        /* Create a timeout limit for the new id by adding the timeout constant*/
        now += self->_timeout;
        /* Generate a id*/
        generate_id(session_id);
        /* Look in the session Map to se if the id is unique*/
    }while(map_find(self->_sessions,session_id) != NULL && cnt < SESSION_MAX_TRY);
    /* When we get here we probably have a unique id or
     * else we accept the id anyway.
     */
    /* Put the id in the session Map*/
    _ltoa((long)now,now_buf,10);
    map_add(self->_sessions,session_id,now_buf);
}

/*
 * Private method that removes the session id if it has timed out.
 */
static int remove_invalid_session_id(Sessionhandler *self, char *session_id)
{
    Mapelement *e;
    time_t now;
    char now_buf[20];
    int exist = 0;

    assert(self);
    assert(session_id);

    /* Get the current time to compare the sessions time with*/
    time ( &now );
    _ltoa((long)now,now_buf,10);
    /* Se if the session id exist*/
    e = map_find(self->_sessions,session_id);
    if (e != NULL)
    {
        exist = 1;
        /* Test if the session has timed out*/
        if (strcmp(e->value,now_buf) < 0)
        {
            /* Remove the timed out session id*/
            map_remove(self->_sessions,e->key);
            /* Return removed*/
            return 1;
        }
    }
    /* se if session id still is alive*/
    if (exist)
    {
        /* Return not removed*/
        return 0;
    }
    else
    {
        /* Return removed*/
        return 1;
    }
}

/*
 * Private method that updates a session id with the current time.
 */
static void update_session_id_time(Sessionhandler *self, char *session_id)
{
    time_t now;
    time_t then;
    char then_buf[20];

    assert(self);
    assert(session_id);

    /* Get current time*/
    time ( &now );
    then = now + self->_timeout;
    _ltoa((long)then,then_buf,10);
    /* Update or create session id with current time*/
    map_set(self->_sessions,session_id,then_buf);
}
```

## sessionhandler.c (continued)

```
/*
 * Method that removes all sessions that has timed out.
 */
void sessionhandler_delete_old(Sessionhandler *self)
{
    Mapelement *e = NULL;
    time_t      now = 0;
    time_t      then = 0;

    /*
     * This is a common function called by several threads so
     * it needs to be protected by a system critical section.
     */
    EnterCriticalSection(&self->_critical_section);
    assert(self);
    /* Get current time*/
    time ( &now );
    assert(now);
    /* Get first session id from map*/
    e = map_findfirst(self->_sessions);
    if (e != NULL)
    {
        /* Loop through all sessions to se if they have timed out*/
        do
        {
            /* Get current sessions timeout limit*/
            then = atol(e->value);
            assert(then);
            /* Compare the sessions timeout limit to current time*/
            if (then - now < 0)
            {
                /* Remove the timed out session*/
                map_remove(self->_sessions,e->key);
            }
            /* Get next session id*/
            e = map_findnext(self->_sessions);
        }while(e != NULL);
    }
    /* End of critical section*/
    LeaveCriticalSection(&self->_critical_section);
}

/*
 * Method that fully generates or updates a session id.
 * The function may generate a new id if the old has timed out.
 */
int sessionhandler_get_valid_id(Sessionhandler *self, char *session_id)
{
    int removed;

    assert(self);
    assert(session_id);

    /*
     * This is a common function called by several server threads so
     * it needs to be protected by a system critical section.
     */
    EnterCriticalSection(&self->_critical_section);

    /* If incomming session id is empty...*/
    if (strlen(session_id) == 0)
    {
        /* ...we generate a new one*/
        get_new_session_id(self,session_id);
        LeaveCriticalSection(&self->_critical_section);
        /* Return that a new id is created*/
        return 1;
    }
    /* Else we have to check the incomming id*/
    else
    {
        /* Test if session id is timed out*/
        removed = remove_invalid_session_id(self,session_id);
        /* Test if it was removed from list*/
        if (removed)
        {
            /* Generate a new session id*/
            get_new_session_id(self,session_id);
            LeaveCriticalSection(&self->_critical_section);
            /* Return that old id was timed out*/
            return 2;
        }
    }
}
```

## sessionhandler.c (continued)

```
    }
    else
    {
        /* Update incomming id with a new timestamp*/
        update_session_id_time(self,session_id);
        LeaveCriticalSection(&self->_critical_section);
        /* Resturn that incomming is was valid*/
        return 0;
    }
}
```

## clientcounter.h

```
/*
 * Declaration of the Clientcounter class.
 * This class is responsible counting the number
 * of clients that is connected to the sever.
 */

#ifndef _CLIENTCOUNTER
#define _CLIENTCOUNTER

#include <windows.h>
#include "common.h"

/*
 * The Clientcounter data type.
 */
typedef struct Clientcounter
{
    int          _count;
    int          _max_clients;
    CRITICAL_SECTION _critical_section;
} Clientcounter;

/* The Clientcounter methods*/
Clientcounter *clientcounter_create(int max_clients);
void clientcounter_delete(Clientcounter *self);
int  clientcounter_hold_client(Clientcounter *self);
void clientcounter_release_client(Clientcounter *self);

#endif /*_CLIENTCOUNTER*/
```

## clientcounter.c

```
/*
 * Definition of the Clientcounter class
 */
#define _CRT_SECURE_NO_WARNINGS
#include "clientcounter.h"
#include "logger.h"

/*
 * Constructor
 */
Clientcounter *clientcounter_create(int max_clients)
{
    Clientcounter *new_obj;

    new_obj = malloc(sizeof(Clientcounter));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->_count = 0;
        new_obj->_max_clients = max_clients;
        InitializeCriticalSection(&new_obj->_critical_section);
    }
    return new_obj;
}

/*
 * Destructor
 */
void clientcounter_delete(Clientcounter *self)
{
    assert(self);
    if (self)
    {
        DeleteCriticalSection(&self->_critical_section);
        free(self);
    }
}

/*
 * This method decides if a new client can connect
 */
int clientcounter_hold_client(Clientcounter *self)
{
    int res;

    /* Take the mutex*/
    EnterCriticalSection(&self->_critical_section);
    /*Check that we dont' create to many simultaneous threads*/
    if (self->_count < self->_max_clients)
    {
        /*Increase number of clienthandler threads*/
        self->_count++;
        res = 1;
    }
    else
    {
        res = 0;
        logger_print(g_logger, 1, "Client rejected");
    }
    /* Give the mutex*/
    LeaveCriticalSection(&self->_critical_section);
    return res;
}

/*
 * This method is used to free a clients place
 */
void clientcounter_release_client(Clientcounter *self)
{
    EnterCriticalSection(&self->_critical_section);
    self->_count--;
    LeaveCriticalSection(&self->_critical_section);
}
```



## directorybrowser.h

```
/*
 * Declaration of the Directorybrowser class.
 * This class is responsible for deciding if a
 * directory content is allowed to be listed.
 * It does that by reading the browser config file
 * i the directory. It does not perform the browsing itself.
 */

#ifndef _DIRECTORYBROWSER
#define _DIRECTORYBROWSER

#include "common.h"

/*
 * The Directorybrowser data type.
 */
typedef struct Directorybrowser
{
    char _directory[FILE_SIZE];
    int _dir_brows_default;
    char *_output;
    char *_pos;
    int _pages;
    int _is_root;
    char *_web_file;
} Directorybrowser;

/* The Directorybrowser methods*/
Directorybrowser *directorybrowser_create(char * directory,int dir_brows_default,char *web_file);
int directorybrowser_can_brows(Directorybrowser *self);
Result directorybrowser_get_brows_data(Directorybrowser *self, char **output);
void directorybrowser_delete(Directorybrowser *self);

#endif /* _DIRECTORYBROWSER*/
```

## directorybrowser.c

```
/*
 * Definition of the Directorybrowser class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "directorybrowser.h"
#include "logger.h"

/*
 * Constructor
 */
Directorybrowser *directorybrowser_create(char * directory,int dir_brows_default,char *web_file)
{
    Directorybrowser *new_obj;

    new_obj = malloc(sizeof(Directorybrowser));
    assert(new_obj);
    if (new_obj)
    {
        strcpy(new_obj->_directory,directory);
        new_obj->_dir_brows_default = dir_brows_default;
        new_obj->_output = 0;
        new_obj->_pos = 0;
        new_obj->_pages = 2;
        new_obj->_web_file = web_file;
        if (!strcmp(web_file,("/") || !strcmp(web_file,"\\\"))
        {
            new_obj->_is_root = 1;
        }
        else
        {
            new_obj->_is_root = 0;
        }
    }
    return new_obj;
}

/*
 * Destructor
 */
void directorybrowser_delete(Directorybrowser *self)
{
    assert(self);
    if (self)
    {
        if (self->_output != 0)
            free(self->_output);
        free(self);
    }
}

int directorybrowser_can_brows(Directorybrowser *self)
{
    FILE *f;
    char dir_file[FILE_SIZE];

    if (!self->_dir_brows_default)
    {
        strcpy(dir_file,self->_directory);
        strcat(dir_file,DIRECTORY_BROWSING_FILE);
        f = NULL;
        /* Try to open the file*/
        f = fopen(dir_file,"rb");
        if (!f)
            return 0;
        else
        {
            /* Could open file so close it*/
            fclose(f);
            return 1;
        }
    }
    return 1;
}

static void write_data(Directorybrowser *self, char *data)
{

```

## directorybrowser.c (continued)

```
char *p;

if( ((self->_pages * DIR_BUFFER_SIZE) - (unsigned int)(self->_pos - self->_output)) < (unsigned
int)strlen(data) + 1)
{
    self->_pages++;
    self->_output = realloc(self->_output,DIR_BUFFER_SIZE * self->_pages);
}

for (p = data; *p != 0; p++)
{
    *(self->_pos)++ = *p;
}
*(self->_pos) = 0;
}

static void print_dir(Directorybrowser *self,char *name,int is_root)
{
    if (self->_is_root)
        if(!strcmp(name,".."))
            return;

    if (strcmp(name,"."))
    {
        write_data(self,"<a href='");
        write_data(self,name);
        write_data(self,"/'>");
        write_data(self,name);
        write_data(self,"</a>");
        write_data(self,"<BR>");
    }
}

static void print_file(Directorybrowser *self,char *name)
{
    if (strcmp(name,DIRECTORY_BROWSING_FILE))
    {
        write_data(self,"<a href='");
        write_data(self,name);
        write_data(self,"'>");
        write_data(self,name);
        write_data(self,"</a>");
        write_data(self,"<BR>");
    }
}
```

Result **directorybrowser\_get\_brows\_data**(Directorybrowser \*self, char \*\*output)

```
{
    WIN32_FIND_DATA    file_data;
    HANDLE             files;
    int                is_root = 0;

    /*Initialize write buffer*/
    if (self->_output == 0)
    {
        self->_output = malloc(DIR_BUFFER_SIZE * self->_pages);
        self->_pos = self->_output;
        self->_pos = self->_pos;
    }

    write_data(self,"<html><head></head><body>");
    write_data(self,"<h2>");
    write_data(self,self->_web_file);
    write_data(self,"</h2>");

    /*Subdirectories*/
    strcat(self->_directory,"*.");
    files = FindFirstFileA(self->_directory,(LPWIN32_FIND_DATA)&file_data);
    if (!files)
        return ERR;
    if (files != INVALID_HANDLE_VALUE)
    {
        if (file_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
            print_dir(self,(char*)file_data.cFileName,is_root);
        while (FindNextFileA(files,(LPWIN32_FIND_DATA)&file_data))
        {
            if (file_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
                print_dir(self,(char*)file_data.cFileName,is_root);
        }
    }
}
```

## directorybrowser.c (continued)

```
    }
    FindClose(files);

    /*Files*/
    files = FindFirstFileA(self->_directory, (LPWIN32_FIND_DATA)&file_data);
    if (!files)
        return ERR;
    if (files != INVALID_HANDLE_VALUE)
    {
        if (!(file_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
            print_file(self, (char*)file_data.cFileName);
        while (FindNextFileA(files, (LPWIN32_FIND_DATA)&file_data))
        {
            if (!(file_data.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
                print_file(self, (char*)file_data.cFileName);
        }
    }
    FindClose(files);

    write_data(self, "</body></html>");
    *output = self->_output;
    return OK;
}
```

## map.h

```
/*
 * Declaration of the Map class.
 * Map is a class to handle lists of a key and
 * a value pairs.
 */

#ifndef _MAP
#define _MAP

/*
 * The Mapelement data type that represents
 * an element in the Map.
 */
typedef struct Mapelement Mapelement;
typedef struct Mapelement
{
    char *key;
    char *value;
    Mapelement *next;
} Mapelement;

/*
 * The Map data type.
 */
typedef struct Map
{
    Mapelement *findPrev;
    Mapelement *current;
    Mapelement *list;
} Map;

/* The Map methods*/
Map *map_create();
void map_delete(Map *self);
Mapelement *map_add(Map *self, char *key, char *value);
void map_remove(Map *self, char *key);
Mapelement *map_find(Map *self, char *key);
Mapelement *map_set(Map *self, char *key, char *value);
char *map_getval(Map *self, char *key);
Mapelement *map_findfirst(Map *self);
Mapelement *map_findnext(Map *self);
void map_set_element_value(Mapelement *e, char *value);

#endif /* _MAP*/
```

## map.c

```
/*
 * Definition of the Map class
 */
#define _CRT_SECURE_NO_WARNINGS
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include "map.h"
#include "logger.h"

/*
 * Constructor
 */
Map *map_create()
{
    Map *new_obj;
    new_obj = malloc(sizeof(Map));
    assert(new_obj);
    if (new_obj)
    {
        new_obj->findPrev = NULL;
        new_obj->list = NULL;
    }
    return new_obj;
}

/*
 * Destructor
 */
void map_delete(Map *self)
{
    Mapelement *e = self->list;
    Mapelement *f;

    assert(self);
    if (self)
    {
        while(e != NULL)
        {
            f = e->next;
            assert(e->key);
            free(e->key);
            assert(e->value);
            free(e->value);
            assert(e);
            free(e);
            e = f;
        }
        free(self);
    }
}

/*
 * Adds a new element to the table, no check if it already exists in the table.
 */
Mapelement *map_add(Map *self, char *key, char *value)
{
    Mapelement *e = NULL, *f;

    assert(self);

    e = malloc(sizeof(Mapelement));
    assert(e);
    if (e == NULL)
        return NULL;

    e->key = malloc(strlen(key) + 1);
    assert(e->key);
    if (e->key == NULL)
    {
        free(e);
        return NULL;
    }

    e->value = malloc(strlen(value) + 1);
    assert(e->value);
    if (e->value == NULL)
    {
        free(e);
        free(e->key);
        return NULL;
    }
}
```

## map.c (continued)

```
    }

    strcpy(e->key, key);
    strcpy(e->value, value);
    e->next = NULL;

    if (self->list == NULL)
    {
        self->list = e;
    }
    else
    {
        f = self->list;
        while(f->next != NULL)
        {
            f = f->next;
        }
        f->next = e;
    }

    return e;
}

/*
 * Removes an element from the map.
 */
void map_remove(Map *self, char *key)
{
    Mapelement *e;
    Mapelement *f;

    assert(self);
    assert(key);

    e = map_find(self, key);

    if (e != NULL)
    {
        /* If first element in list */
        if (e == self->list)
        {
            f = e->next;
            free(e->key);
            free(e->value);
            free(e);
            self->list = f;
            if (e == self->current)
                self->current = f;
        }
        else
        {
            f = e->next;
            free(e->key);
            free(e->value);
            free(e);
            self->findPrev->next = f;
            if (e == self->current)
                self->current = f;
        }
    }
}

/*
 * Finds an element in the map.
 */
Mapelement *map_find(Map *self, char *key)
{
    Mapelement *e;

    assert(self);
    assert(key);

    e = self->findPrev = self->list;

    if (e == NULL)
        return NULL;

    while(e != NULL)
    {
        if (_stricmp(e->key, key) == 0)
```

## map.c (continued)

```
    {
        return e;
    }
    self->findPrev = e;
    e = e->next;
}
return NULL;
}

/*
 * Finds an element in the map and return its value.
 */
char *map_getval(Map *self, char *key)
{
    Mapelement *e;

    assert(self);

    e = map_find(self, key);
    if (e != NULL)
        return e->value;
    else
        return NULL;
}

/*
 * Sets a value to an mapelement.
 * If it exists it is overwritten else updated.
 */
Mapelement *map_set(Map *self, char *key, char *value)
{
    assert(self);

    if (map_find(self, key) != NULL)
    {
        map_remove(self, key);
    }
    return map_add(self, key, value);
}

/*
 * Gets first element in map. Should be called before map_findnext.
 */
Mapelement *map_findfirst(Map *self)
{
    assert(self);
    self->current = self->list;
    return self->current;
}

/*
 * Gets the next element in map. Used to iterate through the maps elements.
 */
Mapelement *map_findnext(Map *self)
{
    assert(self);
    if (self->list != NULL)
    {
        if (self->current->next != NULL)
        {
            self->current = self->current->next;
            return self->current;
        }
    }
    return NULL;
}

void map_set_element_value(Mapelement *e, char *value)
{
    e->value = realloc(e->value, strlen(value)*sizeof(char)+1);
    strcpy(e->value, value);
}
```



## plugin.h

```
/*
 * Declaration of the Plugin API functions.
 */

#ifndef _PLUGIN
#define _PLUGIN

#include <windows.h>

/*
 * The function typedefinitions.
 */

typedef char *(*GET_VERSION)();
typedef char *(*GET_BUILD)();
typedef void (*HANDLE_REQUEST)(SOCKET s, struct sockaddr_in addr);

#endif /*_PLUGIN*/
```

## acceptorplugin.h

```
/*
 * Declaration of the Acceptorpluginplugin class.
 * This class is responsible for connecting the
 * clients to the sever for plugin implemetations.
 */

#ifndef _ACCEPTORPLUGIN
#define _ACCEPTORPLUGIN

#include "thread.h"
#include "plugin.h"

/*
 * The Acceptorpluginplugin data type.
 * Acceptorplugin is a thread class that includes
 * a Thread instance.
 */
typedef struct Acceptorplugin
{
    /* Acceptorplugin is a thread so include a thread instance*/
    Thread          _base_thread;
    SOCKET          _srv_socket;
    SOCKET          _sockio;
    struct sockaddr_in _addr;
    int             _addr_len;
    char            *_plugin_module;
    GET_VERSION     _plugin_get_version;
    GET_BUILD       _plugin_get_build;
    HANDLE_REQUEST  _plugin_handle_request;
    HINSTANCE       _module_instance;
} Acceptorplugin;

/* The Acceptorplugin methods*/
Acceptorplugin *acceptorplugin_create(SOCKET s,char *plugin_name);
void acceptorplugin_delete(Acceptorplugin *self);
static void acceptorplugin_run(void* self);

#endif /*_ACCEPTORPLUGIN*/
```

## acceptorplugin.c

```
/*
 * Definition of the Acceptorplugin class
 */
#define _CRT_SECURE_NO_WARNINGS
#include "acceptorplugin.h"
#include "clienthandlerplugin.h"
#include "logger.h"

/*
 * Constructor
 */
Acceptorplugin *acceptorplugin_create(SOCKET s, char *plugin_name)
{
    Acceptorplugin *new_obj = NULL;

    if (plugin_name != NULL)
    {
        new_obj = malloc(sizeof(Acceptorplugin));
        assert(new_obj);
        if (new_obj)
        {
            new_obj->_plugin_module = malloc(strlen(plugin_name)+10);
            strcpy(new_obj->_plugin_module, plugin_name);
            strcat(new_obj->_plugin_module, ".DLL");
            new_obj->_base_thread._run = acceptorplugin_run;
            new_obj->_base_thread._delete = acceptorplugin_delete;
            new_obj->_srv_socket = s;
            new_obj->_addr_len = sizeof(new_obj->_addr);
            /*Load the plugin dll into memory*/
            new_obj->_module_instance = LoadLibraryA(new_obj->_plugin_module);
            if (new_obj->_module_instance == NULL)
            {
                logger_print(g_logger, 1, "Failed to load plugin module %s", new_obj->_plugin_module);
                return NULL;
            }
            new_obj->_plugin_get_version = (GET_VERSION)GetProcAddress(new_obj->_module_instance, "get_version");
            if (new_obj->_plugin_get_version == NULL)
            {
                logger_print(g_logger, 1, "Invalid plugin module %s", new_obj->_plugin_module);
                return NULL;
            }
            new_obj->_plugin_get_build = (GET_BUILD)GetProcAddress(new_obj->_module_instance, "get_build");
            if (new_obj->_plugin_get_build == NULL)
            {
                logger_print(g_logger, 1, "Invalid plugin module %s", new_obj->_plugin_module);
                return NULL;
            }
            new_obj->_plugin_handle_request = (HANDLE_REQUEST)GetProcAddress(new_obj->_module_instance, "handle_request");
            if (new_obj->_plugin_handle_request == NULL)
            {
                logger_print(g_logger, 1, "Invalid plugin module %s", new_obj->_plugin_module);
                return NULL;
            }
            logger_print(g_logger, 1, "%s (%s build %s) loaded", new_obj->_plugin_module, new_obj->_plugin_get_version(), new_obj->_plugin_get_build());
            thread_start(&new_obj->_base_thread);
        }
    }
    return new_obj;
}

/*
 * Destructor
 */
void acceptorplugin_delete(Acceptorplugin *self)
{
    assert(self);
    if (self)
    {
        if (self->_module_instance)
        {
            FreeLibrary(self->_module_instance);
        }
        shutdown(self->_sockio, 2);
        closesocket(self->_sockio);
        CloseHandle(self->_base_thread._handle);
        free(self->_plugin_module);
    }
}
```

## acceptorplugin.c (continued)

```
        free(self);
    }
}

/*
 * Thread function of Acceptorplugin that performs
 * the real work.
 */
static void acceptorplugin_run(void* self)
{
    Clienthandlerplugin *ch;
    Acceptorplugin      *a = (Acceptorplugin*)self;

    assert(a);
    if (a)
    {
        while (1)
        {
            /* Wait for a client to connect to the server socket*/
            a->_sockio = accept(a->_srv_socket, (struct sockaddr*)&a->_addr, &a->_addr_len);

            /* When a client connects create a client handler plugin
             * thread and let it handle the request.
             * Let the client handler run for it self. It will be deleted
             * when it ends.
             */
            ch = clienthandlerplugin_create(a->_sockio, a->_addr, a->_plugin_handle_request);
            if (!ch)
                logger_print(g_logger, 1, "Error creating clienthandlerplugin");
            /* Return to the top of the loop and handle the next incoming request*/
        }
    }
}
```

## clienthandlerplugin.h

```
/*
 * Declaration of the Clienthandlerplugin class.
 * This class handles a client request from start
 * to the end. It runs in a separate thred.
 */

#ifndef _CLIENTHANDLERPLUGIN
#define _CLIENTHANDLERPLUGIN

#include "thread.h"
#include "common.h"
#include "request.h"
#include "response.h"
#include "plugin.h"

/*
 * The Clienthandlerpluginplugin data type.
 * Clienthandlerplugin is a thread class that includes
 * a Thread instance.
 */
typedef struct Clienthandlerplugin
{
    /* Clienthandlerplugin is a thread so include a thread instance*/
    Thread    _base_thread;
    SOCKET    _sockio;
    struct    sockaddr_in _addr;
    HANDLE_REQUEST _handle_request;
} Clienthandlerplugin;

/* The Clienthandlerplugin methods*/
Clienthandlerplugin *clienthandlerplugin_create(SOCKET s, struct sockaddr_in addr, HANDLE_REQUEST
func);
void clienthandlerplugin_delete(Clienthandlerplugin *self);
static void clienthandlerplugin_run(void* self);

#endif /*_CLIENTHANDLERPLUGIN*/
```

## clienthandlerplugin.c

```
/*
 * Definition of the Clienthandlerplugin class
 */
#define _CRT_SECURE_NO_WARNINGS
#include "clienthandlerplugin.h"
#include "request.h"
#include "response.h"
#include "logger.h"
#include "clientcounter.h"

/* Global Clientcounter object*/
extern Clientcounter *g_clientcounter;

/*
 * Constructor
 */
Clienthandlerplugin *clienthandlerplugin_create(SOCKET s, struct sockaddr_in addr, HANDLE_REQUEST
func)
{
    Clienthandlerplugin *new_obj = NULL;

    if (clientcounter_hold_client(g_clientcounter))
    {
        /* Allocate memory for this object*/
        new_obj = malloc(sizeof(Clienthandlerplugin));
        assert(new_obj);
        if (new_obj)
        {
            new_obj->_base_thread._run = clienthandlerplugin_run;
            /*Set virtual destructor to the destructor of this class*/
            new_obj->_base_thread._delete = clienthandlerplugin_delete;
            new_obj->_sockio = s;
            new_obj->_addr = addr;
            new_obj->_handle_request = func;
            /*Start the work of the thread for this object*/
            thread_start(&new_obj->_base_thread);
        }
    }

    /*If failed to create a clienthandler we close the socket*/
    if (new_obj == NULL)
    {
        /* Close the socket for this object*/
        shutdown (s,2);
        closesocket(s);
    }

    return new_obj;
}

/*
 * Destructor
 */
void clienthandlerplugin_delete(Clienthandlerplugin *self)
{
    assert(self);
    if (self)
    {
        /* Close the socket for this object*/
        shutdown (self->_sockio,2);
        closesocket(self->_sockio);
        /* Close the handle to the thread*/
        CloseHandle(self->_base_thread._handle);
        /* Free memory for this object*/
        free(self);
        /*Count down number of simultanous clienthandler threads*/
        clientcounter_release_client(g_clientcounter);
    }
}

/*
 * Thread function of Clienthandlerplugin
 */
static void clienthandlerplugin_run(void* self)
{
    Clienthandlerplugin *me = (Clienthandlerplugin*)self;

    me->_handle_request(me->_sockio,me->_addr);
}
```

## **minalic.h**

```
/*  
 * Function prototype declarations of the  
 * minaliclib api.  
 */  
  
void echo(char *txt);  
  
void echoex(char *txt,int level);  
  
char *get_config_value(char *key, char *value);
```