LOG IN

Username

Password

Remember me

Sign In

Create an account (https://erlangcentral.org/login/?action=register)

Login using:

(https://erlangcentral.org/login/?action=wordpress_social_authenticate&mode

(https://erlangcentral.org/login/?action=wordpress_social_authenticate&mode

(https://erlangcentral.org/login/?action=wordpress_social_authenticate&mode

(https://erlangcentral.org/login/?action=wordpress_social_authenticate&mode

A Fast Web Server Demonstrating Some Undocumented Erlang Features

From ErlangCentral Wiki

Contents

Author

Sean

Overview

This HOWTO describes a web server written for the day when even Yaws (https://erlangcentral.org/frame/?href=http%3A%2F%2Fyaws.hyber.org) is not quick enough.

The web server presented is quite simple. Even so it is split into 5 modules. Some of these are dictated by the OTP framework, and others are split out for convenience. The 5 modules are:

iserve - API for managing URIs and callbacks
iserve_app - OTP Application behaviour
iserve_sup - OTP Supervisor
iserve_server - Gen_server to own the listening socket and create connections
iserve_socket - Process to handle a single HTTP connection for its lifetime

This HOWTO presents code and descriptions for each of these as they arise.

TCP Server Framework

A web server needs to support lots of connections, so at it's heart it needs to be a multiple connection TCP/IP server. There are any number of ways to arrange a set of erlang processes into such a thing. My favourite method is to have a single gen_server which opens and owns the listen socket (the listening process). This spawns another process which waits in accept until a connection attempt is received. At this time this accepting process sends a message back to the listening process and goes on to handle the traffic. This avoids the need for gen_tcp:controlling_process/2 and associated complexity.

On receipt of the message from the accepting process the listening process spawns a new accepting process and so on.

The listening process also traps exits, and if it receives a non normal exit from the current accepting process it creates a new one. In this way the listening process supervises its acceptor.

Common Header File

The web server creates a #req{} record as it processes each request. This is used as part of the API into implementation callbacks and by the iserve_socket process. Here are the contents of iserve.hrl up front to get it out of the way:

```
% This record characterises the connection from the browser to our server
% it is intended to be a consistent view derived from a bunch of different headers
-record(req, {connection=keep_alive,            % keep_alive | close
          content_length,                   % Integer
          vsn,                              % {Maj,Min}
          method,                           % 'GET'|'POST'
          uri,              % Truncated URI /index.html
          args="",                          % Part of URI after ?
          headers,              % [{Tag, Val}]
          body = <<>>}).            % Content Body
```

Listening Process

Here is the code for the listening process. It is a very basic gen_server which models a single process:

```
-module(iserve_server).

-behaviour(gen_server).

-export([start_link/1, create/2]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
         code_change/3]).

-record(state, {listen_socket,
                port,
                acceptor}).

%%--------------------------------------------------------------------
start_link(Port) when is_integer(Port) ->
    Name = list_to_atom(lists:flatten(io_lib:format("iserve_~w", [Port]))),
    gen_server:start_link({local, Name}, ?MODULE, Port, []).

%% Send message to cause a new acceptor to be created
create(ServerPid, Pid) ->
    gen_server:cast(ServerPid, {create, Pid}).


%% Called by gen_server framework at process startup. Create listening socket
init(Port) ->
    process_flag(trap_exit, true),
    case gen_tcp:listen(Port,[binary, {packet, http},
                              {reuseaddr, true},
                              {active, false},
                              {backlog, 30}]) of
    {ok, Listen_socket} ->
            %%Create first accepting process
        Pid = iserve_socket:start_link(self(), Listen_socket, Port),
        {ok, #state{listen_socket = Listen_socket,
                    port = Port,
            acceptor = Pid}};
    {error, Reason} ->
        {stop, Reason}
    end.


handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

%% Called by gen_server framework when the cast message from create/2 is received
handle_cast({create, _Pid}, #state{listen_socket = Listen_socket} = State) ->
    New_pid = iserve_socket:start_link(self(), Listen_socket, State#state.port),
    {noreply, State#state{acceptor=New_pid}};

handle_cast(_Msg, State) ->
    {noreply, State}.


handle_info({'EXIT', Pid, normal}, #state{acceptor=Pid} = State) ->
    {noreply, State};

%% The current acceptor has died, wait a little and try again
handle_info({'EXIT', Pid, _Abnormal}, #state{acceptor=Pid} = State) ->
    timer:sleep(2000),
    iserve_socket:start_link(self(), State#state.listen_socket, State#state.port),
    {noreply, State};

handle_info(_Info, State) ->
    {noreply, State}.


terminate(Reason, State) ->
    gen_tcp:close(State#state.listen_socket),
    ok.


code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The notable thing about this code is the use of undocumented socket options to set up the initial state of connections made to the web server port.

{backlog, 30} specifies the length of the OS accept queue.
{packet, http} puts the socket into http mode. This makes the socket wait for a HTTP Request line, and if this is received to immediately switch to receiving HTTP header lines. The socket stays in header mode until the end of header marker is received (CR,NL,CR,NL), at which time it goes back to wait for a following HTTP Request line.

Erlang Central *Cafe* ▲

Acceptor/Socket Process

It would be easy enough to create an abstraction of the Listen/Accept process structure and pass in the implementation function as another parameter. For this HOWTO however I'll stick with the most basic model - the acceptor process starts life as an acceptor and goes on to handle the traffic.

The acceptor process is implemented in a separate module iserve_socket. It is in two parts - the first part sets up a bunch of defines and exports and then does the accepting. Here is it is:

```
-module(iserve_socket).

-export([start_link/3]).

-export([init/1]).
-include("iserve.hrl").

-define(not_implemented_501, "HTTP/1.1 501 Not Implemented\r\n\r\n").
-define(forbidden_403, "HTTP/1.1 403 Forbidden\r\n\r\n").
-define(not_found_404, "HTTP/1.1 404 Not Found\r\n\r\n").

-record(c,  {sock,
             port,
             peer_addr,
             peer_port
         }).

```

```
-define(server_idle_timeout, 30*1000).

start_link(ListenPid, ListenSocket, ListenPort) ->
    proc_lib:spawn_link(?MODULE, init, [{ListenPid, ListenSocket, ListenPort}]).

init({Listen_pid, Listen_socket, ListenPort}) ->
    case catch gen_tcp:accept(Listen_socket) of
    {ok, Socket} ->
            %% Send the cast message to the listener process to create a new acceptor
        iserve_server:create(Listen_pid, self()),
        {ok, {Addr, Port}} = inet:peername(Socket),
            C = #c{sock = Socket,
                   port = ListenPort,
                   peer_addr = Addr,
                   peer_port = Port},
        request(C, #req{}); %% Jump to state 'request'
    Else ->
        error_logger:error_report([{application, iserve},
                       "Accept failed error",
                       io_lib:format("~p",[Else])]),
        exit({error, accept_failed})
    end.
```

Note here that the process is started via the proc_lib:spawn_link/3 call. This wraps the normal spawn_link/3 bif so that the same nice error reports are created as for gen_servers, but it allows for a totally unstructured process implementation.

Web Server State Machine

The rest of this module contains the web server code. It is structured as a state machine which follows the state changes of the http socket mode. A single function models each state, and state transitions are simply implemented as a call to the function which owns the next state.

The states are:

request - wait for a HTTP Request line. Transition to state headers if one is received.
headers - collect HTTP headers. After the end of header marker transition to body state.
body - collect the body of the HTTP request if there is one, and lookup and call the implementation callback. Depending on whether the request is persistent transition back to state request to await the next request or exit.

The code for the state request is below. A blocking call is made to gen_tcp:recv/3 with a timeout. The http driver waits for a CRNL terminated line of the form GET / HTTP/1.0. If anything else is received an http_error indication is returned with the erroneous data.

Some broken clients include extra CR or CRNL sequences so these are skipped.

```
request(C, Req) ->
    case gen_tcp:recv(C#c.sock, 0, 30000) of
        {ok, {http_request, Method, Path, Version}} ->
            headers(C, Req#req{vsn = Version,
                               method = Method,
                               uri = Path}, []);
        {error, {http_error, "\r\n"}} ->
        request(C, Req);
    {error, {http_error, "\n"}} ->
            request(C, Req);
    _Other ->
        exit(normal)
    end.
```

The code for the state headers is below. After sending the HTTP request line the http driver automatically switches into header receive mode. The driver looks for values of the form Header-Val: value and sends them one by one after each call to recv.

The driver maintains a hash table of well known header values and if one of those is received from the network it returns the header value as an atom. Otherwise the header value is returned as a string. In both cases the driver takes care of case insensitivity and automatically capitalises the first letter of each hyphen separated word in the header name. The author clearly got a little carried away at this point!

This web server extracts the values of the 'Content-Length' and 'Connection' headers for its own purposes and simply accumulates the other headers in a list to be passed to the application callback.

Erlang Central *Cafe* ▲

At the end of the headers the driver returns {ok, http_eoh}. This is the cue for the web server to skip to body mode. The driver automatically switches to wait for a new request line at this point unless a subsequent call to inet:setops/2 is made.

```
headers(C, Req, H) ->
    case gen_tcp:recv(C#c.sock, 0, ?server_idle_timeout) of
        {ok, {http_header, _, 'Content-Length', _, Val}} ->
            Len = list_to_integer(Val),
            headers(C, Req#req{content_length = Len}, [{'Content-Length', Len}|H]);
        {ok, {http_header, _, 'Connection', _, Val}} ->
            Keep_alive = keep_alive(Req#req.vsn, Val),
            headers(C, Req#req{connection = Keep_alive}, [{'Connection', Val}|H]);
        {ok, {http_header, _, Header, _, Val}} ->
            headers(C, Req, [{Header, Val}|H]);
        {error, {http_error, "\r\n"}} ->
        headers(C, Req, H);
    {error, {http_error, "\n"}} ->
            headers(C, Req, H);
        {ok, http_eoh} ->
            body(C, Req#req{headers = lists:reverse(H)});
    _Other ->
        exit(normal)
    end.

%% Shall we keep the connection alive?
%% Default case for HTTP/1.1 is yes, default for HTTP/1.0 is no.
%% Exercise for the reader - finish this so it does case insensitivity properly !
keep_alive({1,1}, "close")      -> close;
keep_alive({1,1}, "Close")      -> close;
keep_alive({1,1}, _)            -> keep_alive;
keep_alive({1,0}, "Keep-Alive") -> keep_alive;
keep_alive({1,0}, _)            -> close;
keep_alive({0,9}, _)            -> close;
keep_alive(Vsn, KA) ->
    io:format("Got = ~p~n",[{Vsn, KA}]),
    close.
```

The code for the state body is below. At this point we have everything required except the body in the case of a POST request. If present this is retrieved in a single chunk based on the content length supplied. Most web servers will implement some sort of size limit for POST requests. This is still needed in our case to avoid a single client taking all the memory of the Erlang Virtual machine with the subsequent crash. It should be simple to add.

Unless the connection is a keep-alive type the process terminates at the end of processing this function. All resources are cleared up at process exit including open sockets so we do not need to be too careful about explicitly tidying up.

```
body(#c{sock = Sock} = C, Req) ->
    case Req#req.method of
        'GET' ->
            Close = handle_get(C, Req),
            case Close of
                close ->
                    gen_tcp:close(Sock);
                keep_alive ->
                    inet:setopts(Sock, [{packet, http}]),
                    request(C, #req{})
            end;
        'POST' when is_integer(Req#req.content_length) ->
            inet:setopts(Sock, [{packet, raw}]),
            case gen_tcp:recv(Sock, Req#req.content_length, 60000) of
                {ok, Bin} ->
                    Close = handle_post(C, Req#req{body = Bin}),
                    case Close of
                        close ->
                            gen_tcp:close(Sock);
                        keep_alive ->
                            inet:setopts(Sock, [{packet, http}]),
                            request(C, #req{})
                    end;
                _Other ->
                    exit(normal)
            end;
        _Other ->
            send(C, ?not_implemented_501),
            exit(normal)
    end.
```

The rest of the iserve_socket module is below. There is not much left to do. The inet driver has already worked out for us what sort of URI is being used.

The call_mfa/4 function relies on the existence of an ets/mnesia table which converts the URI into a module and function dynamic callback. This must have been created at installation (see section later).

```
handle_get(C, #req{connection = Conn} = Req) ->
    case Req#req.uri of
        {abs_path, Path} ->
            {F, Args} = split_at_q_mark(Path, []),
            call_mfa(F, Args, C, Req),
            Conn;
        {absoluteURI, http, _Host, _, Path} ->
            {F, Args} = split_at_q_mark(Path, []),
            call_mfa(F, Args, C, Req),
            Conn;
        {absoluteURI, _Other_method, _Host, _, _Path} ->
            send(C, ?not_implemented_501),
            close;
        {scheme, _Scheme, _RequestString} ->
            send(C, ?not_implemented_501),
            close;
        _  ->
            send(C, ?forbidden_403),
            close
    end.

handle_post(C, #req{connection = Conn} = Req) ->
    case Req#req.uri of
        {abs_path, Path} ->
            call_mfa(Path, Req#req.body, C, Req),
            Conn;
        {absoluteURI, http, _Host, _, Path} ->
            call_mfa(Path, Req#req.body, C, Req),
            Conn;
        {absoluteURI, _Other_method, _Host, _, _Path} ->
            send(C, ?not_implemented_501),
            close;
        {scheme, _Scheme, _RequestString} ->
            send(C, ?not_implemented_501),
            close;
        _  ->
            send(C, ?forbidden_403),
            close
    end.

call_mfa(F, A, C, Req) ->
    case iserve:lookup(C#c.port, Req#req.method, F) of
        {ok, Mod, Func} ->
            case catch Mod:Func(Req, A) of
                {'EXIT', Reason} ->
```

Erlang Central *Cafe* ▲

```
                    io:format("Worker Crash = ~p~n",[Reason]),
                    exit(normal);
                {200, Headers0, Body} ->
                    Headers = add_content_length(Headers0, Body),
                    Enc_headers = enc_headers(Headers),
                    Resp = [<<"HTTP/1.1 200 OK\r\n">>,
                            Enc_headers,
                            <<"\r\n">>,
                            Body],
                    send(C, Resp)
            end;
        {error, not_found} ->
            send(C, ?not_found_404)
    end.

add_content_length(Headers, Body) ->
    case lists:keysearch('Content-Length', 1, Headers) of
        {value, _} ->
            Headers;
        false ->
            [{'Content-Length', size(Body)}|Headers]
    end.


enc_headers([{Tag, Val}|T]) when is_atom(Tag) ->
    [atom_to_list(Tag), ": ", enc_header_val(Val), "\r\n"|enc_headers(T)];
enc_headers([{Tag, Val}|T]) when is_list(Tag) ->
    [Tag, ": ", enc_header_val(Val), "\r\n"|enc_headers(T)];
enc_headers([]) ->
    [].

enc_header_val(Val) when is_atom(Val) ->
    atom_to_list(Val);
enc_header_val(Val) when is_integer(Val) ->
    integer_to_list(Val);
enc_header_val(Val) ->
    Val.

%% Split the path at the ?. This would have to do all sorts of
%% horrible ../../ path checks and %C3 etc decoding if we wanted to
%% retrieve actual paths to real filesystem files. As it is we only
%% want to look it up as a key in mnesia/ets :)
split_at_q_mark([$?|T], Acc) ->
    {lists:reverse(Acc), T};
split_at_q_mark([H|T], Acc) ->
    split_at_q_mark(T, [H|Acc]);
split_at_q_mark([], Acc) ->
    {lists:reverse(Acc), []}.


send(#c{sock = Sock}, Data) ->
    case gen_tcp:send(Sock, Data) of
        ok ->
            ok;
        _ ->
            exit(normal)
    end.
```

Setting Up The Web Server

The Web server requires two preparation steps. The port number the web server listens on is defined in a file called iserve.conf which must be located in the priv subdirectory of the iserve application. It must contain a line of the form:

{port, 8081}.

If this file is not present then the port number defaults to 8080.

The web server also uses an mnesia table to manage mappings between URLs and implementation callbacks. This may be created and managed with the iserve.erl module:

```
-module(iserve).
-export([create_table/1,
         add_callback/5, delete_callback/3,
         print_callbacks/0,lookup/3]).

-record(iserve_callback, {key,                 % {Port, 'GET'|'POST', Abs_path}
                          mf}).                % {Mod, Func}

create_table(Nodes) ->
    mnesia:create_table(iserve_callback,
                        [{attributes, record_info(fields, iserve_callback)},
                         {disc_copies, Nodes}]).

lookup(Port, Method, Path) ->
    case ets:lookup(iserve_callback, {Port, Method, Path}) of
        [#iserve_callback{mf = {Mod, Func}}] ->
            {ok, Mod, Func};
        [] ->
            {error, not_found}
    end.

add_callback(Port, Method, Path, Mod, Func) when ((Method == 'GET') or (Method == 'POST') and
                                                  is_list(Path) and is_atom(Mod) and
                                                  is_atom(Func) and is_integer(Port)) ->
    mnesia:dirty_write(iserve_callback, #iserve_callback{key = {Port, Method, Path},
                                                         mf = {Mod, Func}}).


delete_callback(Port, Method, Path) ->
    mnesia:dirty_delete(iserve_callback, {Port, Method, Path}).

```

Erlang Central *Cafe* ▲

```
print_callbacks() ->
    All = mnesia:dirty_match_object(#iserve_callback{_ = '_'}),
    io:format("Port\tMethod\tPath\tModule\tFunction~n"),
    lists:foreach(fun(#iserve_callback{key = {Port, Method, Path},
                                       mf = {Module, Function}}) ->
                          io:format("~p\t~p\t~p\t~p\t~p\r\n",[Port, Method, Path, Module, Functi
                  end, All).
```

iserve:create_table([node()]). must be called once at installation.

All Urls must be stored in this table with a module and function which will create the page. So for example the callback for the document root might be defined with:

iserve:add_callback(8081, 'GET', "/", test_iserve_app, do_get).

The callback for index.html could be:

iserve:add_callback(8081, 'GET', "/index.html", module, function2).

Building A Web Application

The simplest kind of iserve web application would be one to simply return a generated page. A function must be implemented which returns {200, Headers, Body} where Headers is a list of {Header-Atom, Val-String} and Body is a binary. For example:

```
-module(test_iserve_app).
-export([do_get/2]).
-include("iserve.hrl").

do_get(#req{} = Req, Args) ->
    {200, [], <<"<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">
<html>
<head>
  <title>Welcome to iserve</title>
</head>
<body>
  Hello
</body>
</html>">>}.
```

Obviously this is an extremely simple example. This is where you come in!

Supervisor And Application Implementation

The web server only needs a little help to become a full blown OTP application. It needs an application behaviour, a supervisor behaviour, and a .app file.

These are presented below.

The Application:

```
-module(iserve_app).
-behaviour(application).
-export([
     start/2,
     stop/1
        ]).

start(_Type, _StartArgs) ->
    case iserve_sup:start_link() of
    {ok, Pid} ->
        alarm_handler:clear_alarm({application_stopped, iserve}),
        {ok, Pid};
    Error ->
        alarm_handler:set_alarm({{application_stopped, iserve}, []}),
        Error
    end.

stop(_State) ->
    alarm_handler:set_alarm({{application_stopped, iserve}, []}),
    ok.
```

Erlang Central *Cafe* ▲

The Supervisor:

```
-module(iserve_sup).
-behaviour(supervisor).
-export([
     start_link/0,
         init/1
        ]).

-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

init([]) ->
    Port = get_config(),
    Server = {iserve_server, {iserve_server, start_link, [Port]},
         permanent, 2000, worker, [iserve_server]},
    {ok, {{one_for_one, 10, 1}, [Server]}}.

get_config() ->
    case file:consult(filename:join(code:priv_dir(iserve), "iserve.conf")) of
        [{port, Port}] ->
            Port;
        _ ->
            8080
    end.
```

The .app file.

A dependency on sasl is only included because of the calls to set and clear alarms in the application behaviour implementation:

```
{application, iserve,
        [{description, "Web Server"},
         {vsn, "%ISERVE_VSN%"},
         {modules, [    iserve_sup,
            iserve_app,
            iserve_server,
                        iserve_socket
            ]},

         {registered, [ iserve_sup]},
         {applications, [kernel, stdlib, sasl]},
     {mod, {iserve_app, []}}]}.
```

License

The code associated with this HOWTO is available under the BSD License (https://erlangcentral.org/frame/?href=http%3A%2F%2Fwww.opensource.org%2Flicenses%2Fbsd-license.php)

Disclaimer

The undocumented features presented in this HOWTO are undocumented because they are not supported by Ericsson. On the other hand they are used in commercially shipping systems.

Retrieved from ‘http://erlangcentral.org/wiki/index.php?title=A_fast_web_server_demonstrating_some_undocumented_Erlang_features&oldid=37315 (http://erlangcentral.org/wiki/index.php?title=A_fast_web_server_demonstrating_some_undocumented_Erlang_features&oldid=37315)’

Our Sponsors: (https://erlangcentral.org/industrial-erlang-user-group/member-companies/)

GOFACTORY (http://go-factory.com/)

Follow us:

(http://www.facebook.com/pages/Erlang-Central/174661552676546)

(https://twitter.com/ErlangCentral)

(https://erlangcentral.org/feed/rss/)

Terms of use and privacy policy (https://erlangcentral.org/terms-of-use-privacy-policy/) | About Us (https://erlangcentral.org/contact/about-us/) | Contact Us (https://erlangcentral.org/contact/contact-us/) | FAQ (https://erlangcentral.org/contact/faq/) | Roadmap (https://erlangcentral.org/contact/erlang-central-roadmap/) | Cafe Help (https://erlangcentral.org/cafe-help/)

Industrial Erlang User Group (https://erlangcentral.org/industrial-erlang-user-group/)