

PLAYER IMPLEMENTATION

The comments in the code should be self-explanatory.

The API consists of a single function – *start_link/3*. It called from *player.sh* script, which is executed for a player process starting. It takes three arguments: IP, Port and Nick, which are then passed to the *start_link/3* function.

Ex.: > ./player.sh "127.0.0.1" 2090 "pawel"

Functions: *init/1*, *handle_call/3*, *handle_cast/2*, *handle_info/2*, *terminate/2*, *code_change/3* are required by a *gen_server* behaviour, which this module implements. For further reading visit http://www.erlang.org/doc/man/gen_server.html.

The *state* record describes the connection settings (IP address, port, socket), game ID, and a buffer for merging incomplete TCP messages. The field *positions* is left for future implementation of actual game state (tics' and tac's positions).

Functions *get_sub_element/2*, *get_attr_value/2*, *msg/1*, *getPlayers/1*, *handle_xml/2*, *thankYouMsg/0*, *leaveGameMsg/0*, *logoutMsg/0*, *ticMsg/0*, *errorMsg/0*, *msgInfo/0* are used specifically for the projects Player implementation. Their purpose and behaviour is described in the code's comments.

```
-module(gamer) .
%-compile(export_all) .

%% API
-export([start_link/3]) .

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

%% for communication testing
-export([testThankYou/0,
        testLeaveGame/0,
        testLogout/0,
        testTic/0,
        testError/0,
        testPlayerLogin/0]) .

-record(state, {address,
               port,
               positions=[],
               buffer = [],
               socket,
               gameId = "5-in-line-tic-tac-toe"
               }).

-include_lib("xmerl/include/xmerl.hrl") .

%%=====
%% API
%%=====

%%-----
%% @doc
%% Starts the server
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
```

```

%% @end
%%-----
start_link(Address, Port, Nick) ->
    gen_server:start_link({local,?MODULE}, ?MODULE, [Address, Port, Nick],
[]).

%%%=====
%%% gen_server callbacks
%%%=====

%%-----
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%-----
init([Address, Port, Nick]) ->
    {ok, Socket} = gen_tcp:connect(Address, Port, [{mode, list}]),
    io:fwrite("Connecting...\n",[]),
    String = io_lib:fwrite("<message type=\"playerLogin\"><playerLogin
nick=\"~s\" gameType=\"5-in-line-tic-tac-toe\"/></message>", [Nick]),
    gen_tcp:send(Socket, String),
    {ok, #state{address=Address, port=Port, socket=Socket, gameId="5-in-
line-tic-tac-toe"}}.

%%-----
%% @private
%% @doc
%% Handling call messages
%%
%% @spec handle_call(Request, From, State) ->
%%                     {reply, Reply, State} |
%%                     {reply, Reply, State, Timeout} |
%%                     {noreply, State} |
%%                     {noreply, State, Timeout} |
%%                     {stop, Reason, Reply, State} |
%%                     {stop, Reason, State}
%% @end
%%-----
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

%%-----
%% @private
%% @doc
%% Handling cast messages. This is used for testing sending messages to a
server.
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%-----
handle_cast(Msg, State) ->
    gen_tcp:send(State#state.socket, Msg),
    {noreply, State}.

```

```

%%-----
%% @private
%% @doc
%% Handling all non call/cast messages. This is used for receiving data over %%
%% TCP.
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                     {noreply, State, Timeout} |
%%                                     {stop, Reason, State}
%% @end
%%-----
handle_info({tcp, _Socket, DataBin}, State) ->
    Data0 = DataBin,
    Data = State#state.buffer++Data0,
    try xmerl_scan:string(Data) of %trying to parse Data as XML
        {Element, Tail} ->
            case handle_xml(Element, State) of
                {ok, State1} -> % the case when player
does not need to answer
                    {noreply, State1#state{buffer=Tail}};
                {ok, State1, Msg} -> % the case when player has to
answer
                    gen_tcp:send(State#state.socket, Msg),
                    {noreply, State1#state{buffer=Tail}};
                    {stop, Reason, Msg} -> % error
                        gen_tcp:send(State#state.socket, Msg),
                        gen_tcp:close(State#state.socket),
                        {stop, Reason, State}
            end
        catch
            ErrType:ErrMsg ->
                io:fwrite("Error parsing: ~p~n", [{ErrType, ErrMsg}]),
                {noreply, State#state{buffer = Data}}
        end,
    {noreply, State};
handle_info({tcp_closed, _Socket}, State) ->
    io:fwrite("TCP connection closed.~n", []),
    {stop, normal, State};
handle_info(Info, State) ->
    {stop, {odd_info, Info}, State}.

%%-----
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the gen_server terminates
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%-----
terminate(Reason, _State) ->
    io:fwrite("Player terminating because of ~p~n.", [Reason]),
    ok.

%%-----
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%-----

```

```

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

%%%=====
%%% Internal functions
%%%=====

%%%=====
%%%XML parsing
%%%=====

%% Gets an XML subelement with tag 'Name' from an element 'El'.
get_sub_element(Name, #xmlElement{} = El) ->
    #xmlElement{content = Content} = El,
    case lists:keyfind(Name, #xmlElement.name, Content) of
        false ->
            false;
        Tuple ->
            Tuple
    end.

%% Gets value of the 'Name' attribute of element 'El'
get_attr_value(Name, #xmlElement{} = El) ->
    #xmlElement{attributes = Attrs} = El,
    case lists:keyfind(Name, #xmlAttribute.name, Attrs) of
        false ->
            false;
        At ->
            At#xmlAttribute.value
    end.

%% Generates an XML element using 'El'
msg(El) ->
    lists:flatten(xmerl:export_simple_content([El], xmerl_xml)).

%% Gets a list of players from an XML element 'List' containing multiple
'Player' tags
getPlayers(List) ->
    [E || {xmlElement,player,_,_,_,_,_,_,_,_,_} = E <- List].

%% Handling incoming messages.
handle_xml(E, State) ->
    case get_attr_value(type, E) of
        "error" ->
            msgInfo(error,State),
            #xmlElement{content=Content} = E,
            [#xmlText{value=Error}] = Content,
            io:fwrite("Received error: ~p~n",[Error]),
            {ok, State};
        "loginResponse" ->
            msgInfo(loginResponse, State),
            El = get_sub_element(response, E),
            Accept = get_attr_value(accept,El),
            case Accept of
                "no" ->
                    io:fwrite("Login denied!~n", []),
                    E2 = get_sub_element(error,E),
                    ErrorId = get_attr_value(id, E2),
                    io:fwrite("Error id = ~p: ", [ErrorId]),
                    case ErrorId of
                        "1" ->
                            io:fwrite("wrong

```

```

nick.~n",[]);

"2" ->
io:fwrite("improper game

type.~n",[]);

"3" ->
io:fwrite("players pool

overflow.~n",[]);

"5" ->
io:fwrite("wrong game

type description data")
end;
"yes" ->
io:fwrite("Login accepted by server ~p!
~n", [State#state.address])
end,
{ok,State};
"gameState" ->
msgInfo(gameState, State),
"5-in-line-tic-tac-toe" =
get_attr_value(id,get_sub_element(gameId, E)),
E2 = get_sub_element(nextPlayer, E),
if E2 == false ->
io:fwrite("Game Over!~n", []),
E3 = get_sub_element(gameOver, E),
#xmlElement{content=Content} = E3,
Players = getPlayers(Content),
Players1 = lists:foldl(fun(Elem, Result)
-> [{get_attr_value(nick,Elem),get_attr_value(result, Elem)}|Result] end,
[],Players),
lists:foreach(fun({Nick,Result}) ->
io:fwrite("Player ~p is a ~p.~n",[Nick,Result]) end, Players1);

true ->
Nick = get_attr_value(nick, E2),
io:fwrite("Next player to move: ~p~n",
[Nick])

end,
E4 = get_sub_element(gameState, E),
E5 = get_sub_element(tic, E4),
if E5 == false ->
E6 = get_sub_element(tac,E4),
X = get_attr_value(x, E6),
Y = get_attr_value(y, E6),
io:fwrite("Last move: tac, x=~p,
y=~p~n", [X,Y]);

true ->
X = get_attr_value(x,E5),
Y = get_attr_value(y, E5),
io:fwrite("Last move: tic, x=~p,
y=~p~n", [X,Y])

end,

{ok, State, thankYouMsg()};
"serverShutdown" ->
msgInfo(serverShutdown,State),
{ok, State};
"championsList" ->
msgInfo(championsList,State),
#xmlElement{content=Content} = E,
Players = getPlayers(Content),
Players1 = lists:foldl(fun(Elem, Result) ->
[{get_attr_value(nick,Elem),get_attr_value(won,
Elem),get_attr_value(lost,Elem)}|Result] end, [], Players),
lists:foreach(fun({Nick,Won,Lost}) -> io:fwrite("Player

```

```

~p: won - ~p, lost - ~p.~n", [Nick, Won, Lost]) end, Players1),
    {ok, State}
end.

%%%=====
%%% Message generation
%%%=====

%% Generates "Thank you" message
thankYouMsg() ->
    Msg = {message, [{type, "thank you"}], [{gameId, [{id, "5-in-line-tic-tac-
toe"}], []}] },
    msg(Msg).

%% Generates "leaveGame" message
leaveGameMsg() ->
    Msg = {message, [{type, "leaveGame"}], [{gameId, [{id, "5-in-line-tic-tac-
toe"}], []}]},
    msg(Msg).

%% Generates "logout" message
logoutMsg() ->
    Msg = {message, [{type, "logout"}], []},
    msg(Msg).

%% Generates "move" message with a 'tic'
ticMsg() ->
    Msg = {message, [{type, "move"}], [
        {gameId, [{id, "5-in-line-tic-tac-toe"}], []},
        {move, [], [{tic, [{x, "1"}, {y, "2"}], []}]}
    ]},
    msg(Msg).

%% Generates exemplary "error" message
errorMsg() ->
    Msg = {message, [{type, "error"}], ["this is a test error"]},
    msg(Msg).

%% Generates exemplary "playerLogin" message - used for testing
playerLoginMsg() ->
    Msg = {message, [{type, "playerLogin"}], [{playerLogin,
[{nick, "pawelMichna"}, {gameType, "5-in-line-tic-tac-toe"}], []}]},
    msg(Msg).

%%%=====
%%% Functions for testing sending messages to a server
%%% =====
testThankYou() ->
    gen_server:cast(gamer, thankYouMsg()).

testLeaveGame() ->
    gen_server:cast(gamer, leaveGameMsg()).

testLogout() ->
    gen_server:cast(gamer, logoutMsg()).

testTic() ->
    gen_server:cast(gamer, ticMsg()).

testError() ->
    gen_server:cast(gamer, errorMsg()).

```

```
testPlayerLogin() ->
    gen_server:cast(gamer, playerLoginMsg()).
```

```
%%=====
%% Helper functions
%%=====
```

```
%% Displays information what message was received and from whom.
msgInfo(Msg,State) ->
    io:fwrite("Received ~p from server ~p, gameId:~p~n",
[Msg,State#state.address,State#state.gameId]).
```