MODULE

xmlrpc − XML−RPC library

DESCRIPTION

This is an HTTP 1.1 compliant XML-RPC Erlang library. It is designed to make it easy to write XML-RPC Erlang clients and/or servers. The library is compliant with the XML-RPC specification published by *http://www.xmlrpc.org/*.

EXPORTS

call(*Socket***,** *URI***,** *Payload***)**
call(*Socket***,** *URI***,** *Payload***)**
call(*Host***,** *Port***,** *URI***,** *Payload***)**
call(*Socket***,** *URI***,** *Payload***,** *KeepAlive***,** *Timeout***)**
call(*Host***,** *Port***,** *URI***,** *Payload***,** *KeepAlive***,** *Timeout***) ->** *Result*

Types Socket = socket()
URI = string()
Payload = {call, Method, [Value]}
Method = atom()
Value = integer() | float() | string() | Boolean | ISO8601Date | Base64 | Struct | Array
Boolean = true | false
ISO8601Date = {date, string()}
Base64 = {base64, string()}
Struct = {struct, [{Key, Value}]}
Key = atom()
Array = {array, [Value]}
Host = string() | ip()
Port = integer()
KeepAlive = true | false
Timeout = integer()
ResponsePayload = {response, [Value]} | {response, Fault}
Fault = {fault, FaultCode, FaultString}
FaultCode = integer()
FaultString = string()
Result = {ok, ResponsePayload} | {error, Reason} | {ok, Socket, ResponsePayload} | {error, Socket, Reason}
Reason = term()

Calls an XML−RPC server listening on *Host*:*Port*. The *URI* and *Payload* is used in the HTTP POST request being sent to the server. The *Value* is converted to XML (see **DATA TYPES** below) and is used as request body.

If *KeepAlive* is **true** a *Socket* is returned. The socket can be used to send several calls on the same connection in accordance with HTTP 1.1. If no server response is received within *Timeout* milli-seconds **{error, timeout}** or **{error,** *Socket***, timeout}** is returned.

KeepAlive and *Timeout* default to **false** and 60000 milli-seconds.

See **EXAMPLES** section below.

start_link(*Handler***)**
start_link(*Port***,** *MaxSessions***,** *Timeout***,** *Handler***,** *State***)**
start_link(*IP***,** *Port***,** *MaxSessions***,** *Timeout***,** *Handler***,** *State***) ->** *Result*

Types Handler = {Module, Function}
Module = Function = atom()
Port = MaxSessions = integer()
Timeout = integer()
State = term()
IP = ip()

Result = {ok, Pid} | {error, Reason}
Pid = pid()
Reason = term()

Starts an XML−RPC server listening on *IP*:*Port*. If no *IP* address is given the server listens on *Port* for all available *IP* addresses. *MaxSessions* is used to restrict the number of concurrent connections. If *MaxSessions* is reached the server accepts no new connections for 5 seconds, i.e. blocking new connect attempts.

Handler is a callback, implemented by *Module*:*Function*/2, which is used to instantiate an XML−RPC server. The *Timeout* value is used if the handler is keepalive oriented. *State* is the initial state given to *Module*:*Function*/2. The resulting *Pid* can be used as input to **xmlrpc:stop/1**.

See **Module:Function/2** and **EXAMPLES** below.

stop(*Pid***) ->** *Result*

Types Pid = pid()
Result = void()

Stops a running XML−RPC server.

*Module***:***Function***(***State***,** *Payload***) ->** *Result*

Types State = term()
Payload = <See above>
Result = {KeepAlive, ResponsePayload} | {KeepAlive, State, Timeout, ResponsePayload}
KeepAlive = true | false
ResponsePayload = <See above>
Timeout = integer()

It is up to you to implement *Function* clauses in *Module* to instantiate an XML−RPC server. Every time an XML-RPC call arrives the *Value* in the *Payload* gets converted to Erlang format and is passed on to *Module*:*Function*/2.

A *Function* clause is supposed to return either a 2-tuple or a 4-tuple. *KeepAlive* **must** be **false** in a 2-tuple and **true** in a 4-tuple. *KeepAlive* decides if the connection to the client should be kept open or not, i.e. compare with the *KeepAlive* argument to **call/{3,4,5,6}** above.

State can be used as a state variable by the callback function and changes made to it is propagated to the next call to *Module*:*Function*/2. The state variable is only meaningful if both the client and the server is keepalive oriented. The *Timeout* specified in **start_link/{1,5,6}** can be updated in the returning 4-tuple.

If *KeepAlive* is **true** and no call arrives within *Timeout* milli-seconds the socket is closed. The socket may be closed by the client before the specified timeout.

See **EXAMPLES** below.

DATA TYPES

The conversion of *Value* in *Payload* and *ResponsePayload* (see above) is done as follows:

```
XML-RPC data type       Erlang data type
-----------------       ----------------
<int>                   integer()
<boolean>               true or false
<string>                string()
<double>                float()
<dateTime.iso8601>      {date, string()}
<struct>                {struct, [{Key, Value}]}
<array>                 {array, [Value]}
<base64>                {base64, string()}
```

Read more about the XML−RPC data types in the XML−RPC specification published by *http://www.xml-rpc.org/*.

Here are some examples on how Erlang format is converted to XML:

42 <int>42</int>

true <boolean>true</boolean>

"Kilroy was here"
<string>Kilroy was here</string>

42.5 <double>42.5</double>

{date, "19980717T14:08:55"}
<dateTime.iso8601>19980717T14:08:55</dateTime.iso8601>

{struct, [{foo, 42}, {bar, 4711}]}

```
<struct>
    <member>
        <name>foo</name><value><int>42</int></value>
    </member>
    <member>
        <name>bar</name><value><int>4711</int></value>
    </member>
</struct>
```

{array, [42, 42.5}

```
<array>
    <data>
        <value><int>42</i4></value>
        <value><double>42.5</double></value>
    </data>
</array>
```

{date, "19980717T14:08:55"}
<dateTime.iso8601>19980717T14:08:55</dateTime.iso8601>

EXAMPLES

You are strongly advised to inspect the example code in the *examples/* directory.

The first example (*fib_server.erl*) calculates Fibonacci values and is a non-keepalive server. The second example (*echo_server.erl*) echoes back any incoming parameters and is a non-keepalive server. The third example (*date_server.erl*) calculates calendar values for given dates and is a keepalive server which uses the state variable to provide login state and different timeout settings. The fourth example (*validator.erl*) is a validation server which can be used to validate the library using the *http://validator.xmlrpc.org/* service.

A snippet from the Fibonacci callback module in the *examples/* directory:

```
handler(_State, {call, fib, [N]}) when integer(N) ->
    {false, {response, [fib(N)]}};
handler(_State, Payload) ->
    FaultString = lists:flatten(io_lib:format("Unknown call: ~p", [Payload])),
    {false, {response, {fault, -1, FaultString}}}.

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1)+fib(N-2).
```

and how it can be called:

```
1> xmlrpc:call({127, 0, 0, 1}, 4567, "/", {call, fib, [0]}).
{ok,{response,[1]}}
```

```
2> xmlrpc:call({127, 0, 0, 1}, 4567, "/", {call, fib, [4]}).
{ok,{response,[5]}}
```

Again: You are strongly advised to inspect the example code in the *examples/* directory.

FILES

http://www.xmlrpc.org/
: Home for the XML−RPC specification.

README
: Main README file for the library.

examples/
: Example code

AUTHOR

Joakim Grebeno − jocke@gleipnir.com