SSH

version 0.9

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.4 Document System.

Contents

SSH Reference Manual

Short Summaries

- Erlang Module **ssh** [page 5] – Main API of the SSH application
- Erlang Module **ssh_cli** [page 6] – SSH Command Line Interface.
- Erlang Module **ssh_cm** [page 8] – SSH connection layer.
- Erlang Module **ssh_sftp** [page 11] – SFTP client.
- Erlang Module **ssh_sftpd** [page 17] – SSH FTP server.
- Erlang Module **ssh_ssh** [page 18] – SSH client.
- Erlang Module **ssh_sshd** [page 19] – SSH server with erlang shell.
- Erlang Module **ssh_transport** [page 20] – SSH transport layer.

ssh

The following functions are exported:

- `start() -> ok | {error, Reason}`
  [page 5] Starts the SSH application
- `stop() -> ok | {error, Reason}`
  [page 5] Stops the SSH application
- `stop`
  [page 5] Stops the SSH application

ssh_cli

The following functions are exported:

- `listen(Shell)`
  [page 6] Start an SSH server with a CLI
- `listen(Shell, Port)`
  [page 6] Start an SSH server with a CLI
- `listen(Shell, Port, Options)`
  [page 6] Start an SSH server with a CLI
- `listen(Shell, Addr, Port, Options)`
  [page 6] Start an SSH server with a CLI
- `stop(Pid) -> ok | {error, Reason}`
  [page 6] Stop the listener

ssh_cm

The following functions are exported:

- connect(Host) -> {ok, Pid} | {error, Error}
  [page 8] Connect to an ssh daemon
- connect(Host, Options) -> {ok, Pid} | {error, Error}
  [page 8] Connect to an ssh daemon
- connect(Host, Port, Options) -> {ok, Pid} | {error, Error}
  [page 8] Connect to an ssh daemon
- listen(UserFun, Options) -> ok
  [page 9] Start an ssh shell
- listen(UserFun, Port, Options) -> ok
  [page 9] Start an ssh shell
- listen(UserFun, Addr, Port, Options) -> ok
  [page 9] Start an ssh shell
- stop_listener(Pid) -> ok | {error, Reason}
  [page 9] Stop the listener

ssh_sftp

The following functions are exported:

- connect(CM) -> {ok, Pid} | {error, Reason}
  [page 11] Connect to an SFTP server
- connect(Host, Options) -> {ok, Pid} | {error, Reason}
  [page 11] Connect to an SFTP server
- connect(Host, Port, Options) -> {ok, Pid} | {error, Reason}
  [page 11] Connect to an SFTP server
- read_file(Server, File) -> {ok, Data} | {error, Reason}
  [page 11] Read a file
- write_file(Server, File, Iolist) -> ok | {error, Reason}
  [page 11] Write a file
- list_dir(Server, Path) -> {ok, Filenames} | {error, Reason}
  [page 12] List directory
- open(Server, File, Mode) -> {ok, Handle} | {error, Reason}
  [page 12] Open a file and return a handle
- opendir(Server, Path) -> {ok, Handle} | {error, Reason}
  [page 12] Open a directory and return a handle
- close(Server, Handle) -> ok | {error, Reason}
  [page 12] Close an open handle
- read(Server, Handle, Len) -> {ok, Data} | eof | {error, Error}
  [page 12] Read from an open file
- pread(Server, Handle, Position, Length) -> {ok, Data} | eof | {error, Error}
  [page 12] Read from an open file
- aread(Server, Handle, Len) -> {async, N} | {error, Error}
  [page 13] Read asynchronously from an open file

- `apread(Server, Handle, Position, Length) -> {async, N} | {error, Error}`
  [page 13] Read asynchronously from an open file
- `write(Server, Handle, Data) -> ok | {error, Error}`
  [page 13] Write to an open file
- `pwrite(Server, Handle, Position, Data) -> ok | {error, Error}`
  [page 13] Write to an open file
- `awrite(Server, Handle, Data) -> ok | {error, Error}`
  [page 13] Write asynchronously to an open file
- `apwrite(Server, Handle, Position, Data) -> ok | {error, Error}`
  [page 13] Write asynchronously to an open file
- `position(Server, Handle, Location) -> {ok, NewPosition | {error, Error}`
  [page 14] Seek position in open file
- `read_file_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`
  [page 14] Get information about a file
- `get_file_info(Server, Handle) -> {ok, FileInfo} | {error, Reason}`
  [page 14] Get information about a file
- `read_link_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`
  [page 14] Get information about a symbolic link
- `write_file_info(Server, Name, Info) -> ok | {error, Reason}`
  [page 15] Write information for a file
- `read_link(Server, Name) -> {ok, Target} | {error, Reason}`
  [page 15] Read symbolic link
- `make_symlink(Server, Name, Target) -> ok | {error, Reason}`
  [page 15] Create symbolic link
- `rename(Server, OldName, NewName) -> ok | {error, Reason}`
  [page 15] Rename a file
- `delete(Server, Name) -> ok | {error, Reason}`
  [page 15] Delete a file
- `make_dir(Server, Name) -> ok | {error, Reason}`
  [page 16] Create a directory
- `del_dir(Server, Name) -> ok | {error, Reason}`
  [page 16] Delete an empty directory
- `stop(Server) -> ok`
  [page 16] Stop sftp session

ssh_sftpd

The following functions are exported:

- `listen(Port) -> {ok, Pid}|{error, Error}`
  [page 17] Starts sftp server
- `listen(Port, Options) -> {ok, Pid}|{error, Error}`
  [page 17] Starts sftp server
- `listen(Addr, Port, Options) -> {ok, Pid}|{error, Error}`
  [page 17] Starts sftp server

ssh_ssh

The following functions are exported:

- connect(Host) -> ok
  [page 18] Start an ssh shell

- connect(Host, Options) -> ok
  [page 18] Start an ssh shell

- connect(Host, Port, Options) -> ok
  [page 18] Start an ssh shell

ssh_sshd

The following functions are exported:

- listen(Port) -> {ok, Pid}|{error, Error}
  [page 19] Connect to an ssh daemon

- listen(Port, Options) -> {ok, Pid}|{error, Error}
  [page 19] Connect to an ssh daemon

- listen(Addr, Port, Options) -> {ok, Pid}|{error, Error}
  [page 19] Connect to an ssh daemon

- stop(Pid) -> ok | {error, Reason}
  [page 19] Stop the listener

ssh_transport

No functions are exported.

SSH

Erlang Module

Interface module for the SSH application

Exports

start() -> ok | {error, Reason}

>   Types:
>
>   • Reason = term()
>
>   Starts the SSH application. Require that the *crypto* application has been started.

stop() -> ok | {error, Reason}
stop

>   Types:
>
>   • Reason = term()
>
>   Stops the SSH application

ssh_cli

Erlang Module

This module implements a CLI (Command Line Interface), for an SSH server. It's used by ssh_sshd to provide an interactive erlang shell as an ssh server.

Since ssh_cli uses the group module, the CLI provides full editing just like in the erlang shell, with history (ctrl-p and ctrl-n), line editing and configurable tab expansion (completion).

A full example of how to use ssh_cli is provided in ssh/examples/ssh_sample_cli.erl.

Exports

listen(Shell)
listen(Shell, Port)
listen(Shell, Port, Options)
listen(Shell, Addr, Port, Options)

> Types:
> - Shell = pid() | fun()
> - Port = integer()
> - Addr = string()
> - Options = [{Option, Value}]
> - Option = atom()
> - Value = term()
>
> Starts a daemon listening on Port. The Shell fun is a function spawning a shell process, containing a read-eval-print-loop using ordinary erlang io (e.g. get_line/1 and fprint).
>
> The daemon's group leader will be connected to the SSH daemon, so that the io will be sent to the remote SSH shell client.
>
> An example of how ssh_cli can be used can be found in ssh/examples/ssh_cli_sample.erl.
>
> The module ssh_sshd is implemented using ssh_cli.
>
> For options, see ssh_cm:listen.

stop(Pid) -> ok | {error, Reason}

> Types:
> - Pid = pid()
> - Reason = atom()

Stops the listener given by `Pid`, existing connections will stay open.

ssh_cm

Erlang Module

This module implements the SSH connection layer.

Exports

connect(Host) -> {ok, Pid} | {error, Error}
connect(Host, Options) -> {ok, Pid} | {error, Error}
connect(Host, Port, Options) -> {ok, Pid} | {error, Error}

> Types:
> - Host = string()
> - Port = integer()
> - Options = [{Option, Value}]
>
> Connects to an SSH server. A gen_server is started and returned if connection is successful, but no channel is started, that is done with session_open/2. The Host is a string with the address of a host running an SSH server. The Port is an integer, the port to connect to. The default is 22, the registered port for SSH.
>
> Options are:
>
> {user_dir, String} Sets the user directory, normally ~/.ssh (containing the files known_hosts, id_rsa<c>, <c>id_dsa, authorized_keys).
>
> {silently_accept_hosts, Boolean} When true, (default is false), hosts are added to the file known_hosts without asking the user.
>
> {user_interaction, Boolean} If true, which is the default, password questions and adding hosts to known_hosts will be asked interactively to the user. (This is done during connection to an SSH server.) If false, both these interactions will throw and the server will not start.
>
> {public_key_alg, ssh_rsa | ssh_dsa} Sets the preferred public key algorithm to use for user authentication. If the the preferred algorithm fails of some reason, the other algorithm is tried. The default is to try ssh_rsa first.
>
> {connect_timeout, Milliseconds | infinity} Sets the default timeout when trying to connect to an SSH server. This timeout will also affect calls later when using the SSH connection.
>
> {user, String} Provide a username. If this option is not given, ssh reads from the environment (LOGNAME or USER on unix, USERNAME on Windows).
>
> {password, String} Provide a password for password authentication. If this option is not given, the user will be asked for a password.
>
> {user_auth, Fun/3} Provide a fun for password authentication. The fun will be called as fun(User, Password, Opts) and should return true or false.

{key␣cb, KeyCallbackModule} Provide a special call-back module for key handling. The call-back module should be modeled after the ssh␣file module. The function that must be exported are: private␣host␣rsa␣key/2, private␣host␣dsa␣key/2, lookup␣host␣key/3 and add␣host␣key/3.

{fd, FD} Allow an existing file-descriptor to be used, given in FD. (Simply passed on to gen␣tcp:listen.)

As usual, boolean options that should be true can be given as an atom instead of a tuple, e.g. silently␣accept␣hosts instead of {silently␣accept␣hosts, true}.

listen(UserFun, Options) -> ok
listen(UserFun, Port, Options) -> ok
listen(UserFun, Addr, Port, Options) -> ok

Types:

- UserFun = fun() -> Pid
- Port = integer()
- Addr = string() | any
- Options = [{Option, Value}]
- Option = atom()
- Value = term()

Starts a server listening for SSH connections on the given port.

UserFun is a function that should spawn and return a server upon incoming connections on the given port (and address).

Port is the port that the server should listen on. Everytime a connection is made on that port, the UserFun is called, and the returned process is used as a user process for the server.

Options are:

{system␣dir, String} Sets the system directory, containing the host files that identifies the host for ssh. The default is /etc/ssh, but note that SSH normally requires the host files there to be readable only by root.

{user␣passwords, [{User, Password}]} Provide passwords for password authentication.They will be used when someone tries to connect to the server and public key user autentication fails. The option provides a list of valid user names and the corresponding password. User and Password are strings.

{password, String} Provide a global password that will authenticate any user (use with caution!).

If neither of these options is given, the server will be unable to authenticate with password.

{pwdfun, fun/2} Provide a function for password validation. This is called with user and password as strings, and should return true if the password is valid and false otherwise.

{fd, FD} Allow an existing file-descriptor to be used, given in FD. (Simply passed on to gen␣tcp:listen.)

stop␣listener(Pid) -> ok | {error, Reason}

Types:

- Pid = pid()
- Reason = atom()

Stops the listener, given by `Pid`, existing connections will stay open.

ssh_sftp

Erlang Module

This module implements an SFTP (SSH FTP) client. SFTP is a secure, encrypted file transfer service available for SSH.

The errors returned are from the SFTP server, and are often not posix error codes.

Exports

connect(CM) -> {ok, Pid} | {error, Reason}
connect(Host, Options) -> {ok, Pid} | {error, Reason}
connect(Host, Port, Options) -> {ok, Pid} | {error, Reason}

> Types:
> - Host = string()
> - CM = pid()
> - Port = integer()
> - Options = [{Option, Value}]
> - Option = atom()
> - Value = term()
> - Reason = term()
>
> Connects to an SFTP server. A gen_server is started and returned if connection is successful. This server is used to perform SFTP commands on the server.
>
> For options, see ssh_cm:connect.

read_file(Server, File) -> {ok, Data} | {error, Reason}

> Types:
> - Server = pid()
> - File = string()
> - Data = binary()
> - Reason = term()
>
> Reads a file from the server, and returns the data in a binary, like file:read_file/1.

write_file(Server, File, Iolist) -> ok | {error, Reason}

> Types:
> - Server = pid()
> - File = string()
> - Data = binary()

- Reason = term()

Writes a file to the server, like `file:write_file/2`. The file is created if it's not there.

`list_dir(Server, Path) -> {ok, Filenames} | {error, Reason}`

Types:
- Server = pid()
- Path = string()
- Filenames = [Filename]
- Filename = string()
- Reason = term()

Lists the given directory on the server, returning the filenames as a list of strings.

`open(Server, File, Mode) -> {ok, Handle} | {error, Reason}`

Types:
- Server = pid()
- File = string()
- Mode = [Modeflag]
- Modeflag = read | write | creat | trunc | append | binary
- Handle = term()
- Reason = term()

Opens a file on the server, and returns a handle that is used for reading or writing.

`opendir(Server, Path) -> {ok, Handle} | {error, Reason}`

Types:
- Server = pid()
- Path = string()
- Reason = term()

Opens a handle to a directory on the server, the handle is used for reading directory contents.

`close(Server, Handle) -> ok | {error, Reason}`

Types:
- Server = pid()
- Handle = term()
- Reason = term()

Closes a handle to an open file or directory on the server.

`read(Server, Handle, Len) -> {ok, Data} | eof | {error, Error}`
`pread(Server, Handle, Position, Length) -> {ok, Data} | eof | {error, Error}`

Types:
- Server = pid()
- Handle = term()
- Position = integer()

- Len = integer()
- Data = string() | binary()
- Reason = term()

Reads `Len` bytes from the file referenced by `Handle`. Returns {ok, `Data`}, or `eof`, or {error, `Reason`}. If the file is opened with `binary`, `Data` is a binary, otherwise it is a string.

If the file is read past eof, only the remaining bytes will be read and returned. If no bytes are read, `eof` is returned.

The `pread` function reads from a specified position, combining the `position` and `read` functions.

`aread(Server, Handle, Len) -> {async, N} | {error, Error}`

`apread(Server, Handle, Position, Length) -> {async, N} | {error, Error}`

Types:
- Server = pid()
- Handle = term()
- Position = integer()
- Len = integer()
- N = term()
- Reason = term()

Reads from an open file, without waiting for the result. If the handle is valid, the function returns {async, `N`}, where N is a term guaranteed to be unique between calls of `aread`. The actual data is sent as a message to the calling process. This message has the form {async_reply, `N`, `Result`}, where `Result` is the result from the read, either {ok, `Data`}, or `eof`, or {error, `Error`}.

The `apread` function reads from a specified position, combining the `position` and `aread` functions.

`write(Server, Handle, Data) -> ok | {error, Error}`

`pwrite(Server, Handle, Position, Data) -> ok | {error, Error}`

Types:
- Server = pid()
- Handle = term()
- Position = integer()
- Data = iolist()
- Reason = term()

Write `data` to the file referenced by `Handle`. The file should be opened with `write` or `append` flag. Returns `ok` if successful and {error, `Reason`} otherwise.

Typical error reasons are:

ebadf The file is not opened for writing.

enospc There is a no space left on the device.

`awrite(Server, Handle, Data) -> ok | {error, Error}`

`apwrite(Server, Handle, Position, Data) -> ok | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Position = integer()
- Len = integer()
- Data = binary()
- Reason = term()

Writes to an open file, without waiting for the result. If the handle is valid, the function returns {async, N}, where N is a term guaranteed to be unique between calls of awrite. The result of the write operation is sent as a message to the calling process. This message has the form {async_reply, N, Result}, where Result is the result from the write, either ok, or {error, Error}.

The apwrite writes on a specified position, combining the position and awrite operations.

position(Server, Handle, Location) -> {ok, NewPosition | {error, Error}

Types:

- Server = pid()
- Handle = term()
- Location = Offset | {bof, Offset} | {cur, Offset} | {eof, Offset} | bof | cur | eof
- Offset = int()
- NewPosition = integer()
- Reason = term()

Sets the file position of the file referenced by Handle. Returns {ok, NewPosition (as an absolute offset) if successful, otherwise {error, Reason}. Location is one of the following:

Offset The same as {bof, Offset}.

{bof, Offset} Absolute offset.

{cur, Offset} Offset from the current position.

{eof, Offset} Offset from the end of file.

bof | cur | eof The same as above with Offset 0.

read_file_info(Server, Name) -> {ok, FileInfo} | {error, Reason}
get_file_info(Server, Handle) -> {ok, FileInfo} | {error, Reason}

Types:

- Server = pid()
- Name = string()
- Handle = term()
- FileInfo = record()
- Reason = term()

Returns a file_info record from the file specified by Name or Handle, like file:read_file_info/2.

read_link_info(Server, Name) -> {ok, FileInfo} | {error, Reason}

Types:

- Server = pid()
- Name = string()
- Handle = term()
- FileInfo = record()
- Reason = term()

Returns a `file_info` record from the symbolic link specified by `Name` or `Handle`, like `file:read_link_info/2`.

`write_file_info(Server, Name, Info) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Info = record()
- Reason = term()

Writes file information from a `file_info` record to the file specified by `Name`, like `file:write_file_info`.

`read_link(Server, Name) -> {ok, Target} | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Target = string()
- Reason = term()

Read the link target from the symbolic link specified by `name`, like `file:read_link/1`.

`make_symlink(Server, Name, Target) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Target = string()
- Reason = term()

Creates a symbolic link pointing to `Target` with the name `Name`, like `file:make_symlink/2`.

`rename(Server, OldName, NewName) -> ok | {error, Reason}`

Types:

- Server = pid()
- OldName = string()
- NewName = string()
- Reason = term()

Renames a file named `OldName`, and gives it the name `NewName`, like `file:rename/2`

`delete(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Deletes the file specified by `Name`, like `file:delete/1`

`make_dir(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Creates a directory specified by `Name`. `Name` should be a full path to a new directory. The directory can only be created in an existing directory.

`del_dir(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Deletes a directory specified by `Name`. The directory should be empty, and

`stop(Server) -> ok`

Types:

- Server = pid()

Stops the `sftp` session, closing the connection. Any open files on the server will be closed.

ssh_sftpd

Erlang Module

This module implements an SFTP server.

Exports

`listen(Port) -> {ok, Pid}|{error, Error}`
`listen(Port, Options) -> {ok, Pid}|{error, Error}`
`listen(Addr, Port, Options) -> {ok, Pid}|{error, Error}`

Types:

- Port = integer()
- Addr = string()
- Options = [{Option, Value}]

Starts an SFTP server on the given port. The server listens for connection of an SFTP client.

Options are:

{`cwd`, `String`} Sets the initial current working directory for the server.

{`file_handler`, `CallbackModule`} Determines which module to call for communicating with the file server. Default value is ssh_sftpd_file that uses the file and filelib API:s to access the standard OTP file server. This option may be used to plug in the use of other file servers.

{`root`, `String`} Sets the sftp root directory. The user will then not be able to see any files above this root. If for instance the root is set to `/tmp` the user will see this directory as / and if the user does cd `/etc` the user will end up in `/tmp/etc`.

For more options, see `ssh_cm:listen`.

ssh_ssh

Erlang Module

This module implements a simple SSH client in erlang, providing an interactive shell to another computer.

Exports

```
connect(Host) -> ok
connect(Host, Options) -> ok
connect(Host, Port, Options) -> ok
```

Types:

- Host = string()
- Port = integer()
- Options = [Option]

`connect` starts an interactive shell to an SSH server on the given `Host`. The function waits for user input, and will not return until the remote shell is ended.(e.g. on `exit` from the shell).

For options, see `ssh_cm:connect`

ssh_sshd

Erlang Module

This module implements an erlang shell as an SSH server.

Exports

```
listen(Port) -> {ok, Pid}|{error, Error}
listen(Port, Options) -> {ok, Pid}|{error, Error}
listen(Addr, Port, Options) -> {ok, Pid}|{error, Error}
```

> Types:
>
> - Addr = string()
> - Port = integer()
> - Options = [{Option, Value}]
>
> Create a listener on the given port. (It calls `ssh_cli:listen` with `shell:start/0` as argument.) An SSH client can be used to connect to the listener and execute erlang commands.
>
> Unix example:
>
> ```
> 1> ssh_sshd:listen(9999, [{system_dir, "."}])
> <0.59.0>
> ```
>
> On a unix shell:
>
> ```
> bash@balin$ ssh -p 9999 balin
> Eshell V5.4.9.1  (abort with ^G)
> 1> exit().
> Connection to balin closed.
> bash@balin$
> ```
>
> This assumes that the current dir contains a private host key.
>
> For options, see `ssh_cli:listen/3` and `ssh_cm:listen/4`.

```
stop(Pid) -> ok | {error, Reason}
```

> Types:
>
> - Pid = pid()
> - Reason = atom()
>
> Stops the listener given by `Pid`.

ssh_transport

Erlang Module

This module implements the SSH connection layer, as described in
`draft-ietf-secsh-transport-24`.

This module should not normally be called by a client application.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.