# Set up an SSH client in 5 minutes

SSHD is designed to easily allow setting up and using an SSH client in a few simple steps. The client needs to be configured and then started before it can be used to connect to an SSH server. There are a few simple steps for creating a client instance - for more details refer to the SshClient class.

## Creating an instance of the SshClient class

This is simply done by calling

SshClient client = SshClient.setupDefaultClient();

The call will create an instance with a default configuration suitable for most use cases - including ciphers, compression, MACs, key exchanges, signatures, etc... If your code requires some special configuration, one can look at the code for setupDefaultClient and checkConfig as a reference for available options and configure the SSH client the way you need.

## Set up client side security

The SSH client contains some security related configuration that one needs to consider

### ServerKeyVerifier

client.setServerKeyVerifier(...); sets up the server key verifier. As part of the SSH connection initialization protocol, the server proves its "identity" by presenting a public key. The client can examine the key (e.g., present it to the user via some UI) and decide whether to trust the server and continue with the connection setup. By default the client is initialized with an AcceptAllServerKeyVerifier that simply logs a warning that an un-verified server key was accepted. There are other out-of-the-box verifiers available in the code:

* RejectAllServerKeyVerifier - rejects all server key - usually used in tests or as a fallback verifier if none of it predecesors validated the server key
* RequiredServerKeyVerifier - accepts only **one** specific server key (similar to certificate pinning for SSL)
* KnownHostsServerKeyVerifier - uses the [known\_hosts](https://en.wikibooks.org/wiki/OpenSSH/Client_Configuration_Files#Public_Keys_from_other_Hosts_.E2.80.93_.7E.2F.ssh.2Fknown_hosts) file to validate the server key. One can use this class + some existing code to **update** the file when new servers are detected and their keys are accepted.

Of course, one can implement the verifier in whatever other manner is suitable for the specific code needs.

### ClientIdentityLoader/KeyPairProvider

One can set up the public/private keys to be used in case a password-less authentication is needed. By default, the client is configured to automatically detect and use the identity files residing in the user's *~/.ssh* folder (e.g., *id\_rsa*, *id\_ecdsa*) and present them as part of the authentication process. **Note:** if the identity files are encrypted via a password, one must configure a FilePasswordProvider so that the code can decrypt them before using and presenting them to the server as part of the authentication process. Reading key files in PEM format (including encrypted ones) is supported by default for the standard keys and formats. Using additional non-standard special features requires that the [Bouncy Castle](https://www.bouncycastle.org/) supporting artifacts be available in the code's classpath.

#### Providing passwords for encrypted key files

The FilePasswordProvider is required for all private key files that are encrypted and being loaded (not just the "identity" ones). If the user knows ahead of time that the file being currently decoded is not encrypted, a *null* provider may be used (if the file turns out to be encrypted though an exception will be thrown in this case).

The FilePasswordProviderhas support for a **retry mechanism** via its handleDecodeAttemptResult. When the code detects an encrypted private key, it will start a loop where it prompts for the password, attempts to decode the key using the provided password and then informs the provider of the outcome - success or failure. If failure is signaled, then the provider can decide whether to retry using a new password, abort (with exception) or ignore. If the provider chooses to ignore the failure, then the code will make a best effort to proceed without the (undecoded) key.

The invoked methods are provided with a NamedResource that provides an indication of the key source "name" that is being attempted. This name can be used in order to prompt the user interactively and provide a useful "hint" as to the password that needs to be provided. Furthermore, the vast majority of the provided NamedResource-s also implement IoResource - which means that the code can find out what type of resource is being attempted - e.g., a file [Path](https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Path.html), a [URL](https://docs.oracle.com/javase/8/docs/api/java/net/URL.html), a [URI](https://docs.oracle.com/javase/8/docs/api/java/net/URI.html), etc. - and modify it's behavior accordingly.

#### OpenSSH file format support

The code supports [OpenSSH](http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/PROTOCOL.key?rev=1.1&content-type=text/x-cvsweb-markup) formatted files without any specific extra artifacts (although for reading *ed25519* keys one needs to add the *EdDSA* support artifacts). For **encrypted** files only the the bcrypt key derivation function (KDF) is [currently supported](https://issues.apache.org/jira/browse/SSHD-708). In this context, the maximum allowed number of rounds has been set to ~255 in order to protect the decryption process from malformed or malicious data. However, since the protocol allows for 2^31 values, it is possible to modify the default by calling BCryptKdfOptions#setMaxAllowedRounds() **programmatically** at any time - please note that

* The setting is **global** - i.e., affects all decryption attempts from then on and not just for the current SSH session or thread.
* The setting value is never allowed to be non-positive - any attempt to set such a value programmatically throws an exception.

The usual *OpenSSH* default seems to be 16, but users can ask for more (or less) by generating an encrypted key via [ssh-keygen -a NNN](http://man7.org/linux/man-pages/man1/ssh-keygen.1.html). However, this comes at a cost:

-a rounds

When saving a private key this option specifies the number of KDF (key derivation function) rounds used. Higher numbers result in slower passphrase verification

Various discussions on the net seem to indicate that 64 is the value at which many computers start to slow down noticeably, so our default limit seems quite suitable (and beyond) for most cases we are likely to encounter "in the wild".

### UserInteraction

This interface is required for full support of keyboard-interactive authentication protocol as described in [RFC 4256](https://www.ietf.org/rfc/rfc4256.txt). The client can handle a simple password request from the server, but if more complex challenge-response interaction is required, then this interface must be provided - including support for SSH\_MSG\_USERAUTH\_PASSWD\_CHANGEREQ as described in [RFC 4252 section 8](https://www.ietf.org/rfc/rfc4252.txt).

While RFC-4256 support is the primary purpose of this interface, it can also be used to retrieve the server's welcome banner as described in [RFC 4252 section 5.4](https://www.ietf.org/rfc/rfc4252.txt) as well as its initial identification string as described in [RFC 4253 section 4.2](https://tools.ietf.org/html/rfc4253#section-4.2).

In this context, regardless of whether such interaction is configured, the default implementation for the client side contains code that attempts to auto-detect a password prompt. If it detects it, then it attempts to use one of the registered passwords (if any) as the interactive response to the server's challenge - (see client-side implementation of UserAuthKeyboardInteractive#useCurrentPassword method). Basically, detection occurs by checking if the server sent **exactly one** challenge with no requested echo, and the challenge string looks like "... password ...:" (**Note:** the auto-detection and password prompt detection patterns are configurable).

## Using the SshClient to connect to a server

Once the SshClient instance is properly configured it needs to be start()-ed in order to connect to a server. **Note:** one can use a single SshClient instance to connnect to multiple servers as well as modifying the default configuration (ciphers, MACs, keys, etc.) on a per-session manner (see more in the *Advanced usage* section). Furthermore, one can change almost any configured SshClient parameter - although its influence on currently established sessions depends on the actual changed configuration. Here is how a typical usage would look like

SshClient client = SshClient.setupDefaultClient();  
 // override any default configuration...  
 client.setSomeConfiguration(...);  
 client.setOtherConfiguration(...);  
 client.start();  
  
 // using the client for multiple sessions...  
 try (ClientSession session = client.connect(user, host, port)  
 .verify(...timeout...)  
 .getSession()) {  
 session.addPasswordIdentity(...password..); // for password-based authentication  
 // or  
 session.addPublicKeyIdentity(...key-pair...); // for password-less authentication  
 // Note: can add BOTH password AND public key identities - depends on the client/server security setup  
  
 session.auth().verify(...timeout...);  
 // start using the session to run commands, do SCP/SFTP, create local/remote port forwarding, etc...  
 }  
  
 // NOTE: this is just an example - one can open multiple concurrent sessions using the same client.  
 // No need to close the previous session before establishing a new one  
 try (ClientSession anotherSession = client.connect(otherUser, otherHost, port)  
 .verify(...timeout...)  
 .getSession()) {  
 anotherSession.addPasswordIdentity(...password..); // for password-based authentication  
 anotherSession.addPublicKeyIdentity(...key-pair...); // for password-less authentication  
 anotherSession.auth().verify(...timeout...);  
 // start using the session to run commands, do SCP/SFTP, create local/remote port forwarding, etc...  
 }  
  
 // exiting in an orderly fashion once the code no longer needs to establish SSH session  
 // NOTE: this can/should be done when the application exits.  
 client.stop();

## Configuring the protocol exchange phase

[RFC 4253 section 4.2](https://tools.ietf.org/html/rfc4253#section-4.2) does not specify when the client/server should send their respective identification strings. All it states is that these strings must be available before KEX stage since they participate in it. By default, the client sends its identification string immediately upon session being established. However, this can be modified so that the client waits for the server's identification before sending its own.

SshClient client = ...setup client...  
 PropertyResolverUtils.updateProperty(  
 client, ClientFactoryManager.SEND\_IMMEDIATE\_IDENTIFICATION, false);  
 client.start();

A similar configuration can be applied to sending the initial SSH\_MSG\_KEXINIT message - i.e., the client can be configured to wait until the server's identification is received before sending the message. This is done in order to allow clients to customize the KEX phase according to the parsed server identification.

SshClient client = ...setup client...  
 PropertyResolverUtils.updateProperty(  
 client, ClientFactoryManager.SEND\_IMMEDIATE\_KEXINIT, false);  
 client.start();

**Note:** if immediate sending of the client's identification is disabled, SSH\_MSG\_KEXINIT message sending is also automatically delayed until after the server's identification is received.

A viable configuration might be to send the client's identification immediately, but delay the client's SSH\_MSG\_KEXINIT message sending until the server's identification is received so that the client can customize the session based on the server's identity. This is a more likely configuration then delaying the client's own identification in order to be able to cope with port multiplexors such as [sslh](http://www.rutschle.net/tech/sslh/README.html). Such multiplexors usually require that the client send an initial packet immediately after connection is established so that they can analyze it and route it to the correct server (*ssh* in this case). If we delay the client's identification, then obviously no server identification will ever be received since the multiplexor does not know how to route the connection.

## Keeping the session alive while no traffic

The client-side implementation supports several mechanisms for maintaining the session alive as far as the **server** is concerned regardless of the user's own traffic:

* Sending SSH\_MSG\_IGNORE messages every once in a while.
* This mechanism is along the lines of [PUTTY null packets configuration](https://patrickmn.com/aside/how-to-keep-alive-ssh-sessions/). It generates small [SSH\_MSG\_IGNORE](https://tools.ietf.org/html/rfc4253#section-11.2) messages. The way to set this mechanism up is via the setSessionHeartbeat API.
* *Note:* the same effect can also be achieved by setting the relevant properties documented in SessionHeartbeatController, but it is highly recommended to use the API - unless one needs to control these properties **externally** via -Dxxx JVM options.
* Sending keepalive@... [global requests](https://tools.ietf.org/html/rfc4254#section-4).
* The feature is controlled via the ClientFactoryManager#HEARTBEAT\_REQUEST and HEARTBEAT\_INTERVAL properties - see the relevant documentation for these features. The simplest way to activate this feature is to set the HEARTBEAT\_INTERVAL property value to the **milliseconds** value of the requested heartbeat interval.
* This configuration only ensures that the **server** does not terminate the session due to no traffic. If the incoming traffic from the server may also suffer from long "quiet" periods, one runs the risk of a **client** time-out. In order to avoid this, it is possible to activate the wantReply option for the global request. This way, there is bound to be some packet response (even if failure - which will be ignored by the heartbeat code). In order to activate this option one needs to set the HEARTBEAT\_REPLY\_WAIT property value to a **positive** value specifying the number of **milliseconds** the client is willing to wait for the server's reply to the global request.
* Customized user code
* In order to support customized user code for this feature, the ReservedSessionMessagesHandler can be used to implement any kind of user-defined heartbeat. *Note:* if the user configured such a mechanism, then the sendReservedHeartbeat method **must** be implemented since the default throws UnsupportedOperationException which will cause the session to be terminated the 1st time the method is invoked.

**Note(s):**

* Mechanisms are disabled by default - they need to be activated explicitly.
* Mechanisms can be activated either on the SshClient (for **global** setup) and/or the ClientSession (for specific session configuration).
* The keepalive@,,,, mechanism **supersedes** the other mechanisms if activated.
  + If specified timeout expires for the wantReply option then session will be **closed**.
  + *Any* response - including [SSH\_MSH\_REQUEST\_FAILURE](https://tools.ietf.org/html/rfc4254#page-4) is considered a "good" response for the heartbeat request. In this context, a special patch has been introduced in [SSHD-968](https://issues.apache.org/jira/browse/SSHD-968) that converts an SSH\_MSG\_UNIMPLEMENTED response to such a global request into a SSH\_MSH\_REQUEST\_FAILURE since some servers have been found that violate the standard and reply with it to the request.
* When using the CLI, these options can be configured using the following -o key=value properties:
  + ClientAliveInterval - if positive the defines the heartbeat interval in **seconds**.
  + ClientAliveUseNullPackets - *true* if use the SSH\_MSG\_IGNORE mechanism, *false* if use global request (default).
  + ClientAliveReplyWait - if positive, then activates the wantReply mechanism and specific the expected response timeout in **seconds**.

## Running a command or opening a shell

When running a command or opening a shell, there is an extra concern regarding the PTY configuration and/or the reported environment variables. By default, unless specific instructions are provided, the code uses some internal defaults - which however, might not be adequate for the specific client/server.

// Assuming one has obtained a ClientSession as already shown  
 try (ClientChannel channel = session.createShellChannel(/\* use internal defaults \*/)) {  
 channel.setIn(...stdin...);  
 channel.setOut(...stdout...);  
 channel.setErr(...stderr...);  
 // ... spawn the thread(s) that will pump the STDIN/OUT/ERR  
 try {  
 channel.open().verify(...some timeout...);  
 // Wait (forever) for the channel to close - signalling shell exited  
 channel.waitFor(EnumSet.of(ClientChannelEvent.CLOSED), 0L);  
 } finally {  
 // ... stop the pumping threads ...  
 }  
 }  
  
 // In order to override the PTY and/or environment  
 Map<String, ?> env = ...some environment...  
 PtyChannelConfiguration ptyConfig = ...some configuration...  
 try (ClientChannel channel = session.createShellChannel(ptyConfig, env)) {  
 ... same code as before ...  
 }  
  
 // the same code can be used when opening a ChannelExec in order to run a single command

One possible source of PTY configuration is code that provides some default initializations based on the detected O/S type - PtyChannelConfigurationMutator#setupSensitiveDefaultPtyConfiguration. Of course, the user may use whatever other considerations when opening such a channel.

**Caveat Emptor:** If the detected O/S type is Unix/Linux, then the setupSensitiveDefaultPtyConfiguration code issues an stty command and parses the results (see SttySupport class). Since this involves using System#exec it is a source of concern as it may hang, throw an exception, provide corrupted data, etc...