# Embedding an SSHD server instance in 5 minutes

SSHD is designed to be easily embedded in your application as an SSH server. The embedded SSH server needs to be configured before it can be started. Essentially, there are a few simple steps for creating the server - for more details refer to the SshServer class.

## Creating an instance of the SshServer class

Creating an instance of SshServer is as simple as creating a new object

SshServer sshd = SshServer.setUpDefaultServer();

It will configure the server with sensible defaults for ciphers, macs, key exchange algorithm, etc... If different behavior is required, one should consult the code of the setUpDefaultServer as well as checkConfig methods as a reference for available options and configure the SSH server the way it is needed.

## Configuring the server instance

There are a few things that need to be configured on the server before being able to actually use it:

* Port - sshd.setPort(22); - sets the listen port for the server instance. If not set explicitly then a **random** free port is selected by the O/S. In any case, once the server is start()-ed one can query the instance as to the assigned port via sshd.getPort().

In this context, the listen bind address can also be specified explicitly via sshd.setHost(...some IP address...) that causes the server to bind to a specific network address rather than **all** addresses (the default). Using "0.0.0.0" as the bind address is also tantamount to binding to all addresses.

* KeyPairProvider - sshd.setKeyPairProvider(...); - sets the host's private keys used for key exchange with clients as well as representing the host's "identities". There are several choices - one can load keys from standard PEM files or generate them in the code. It's usually a good idea to save generated keys, so that if the SSHD server is restarted, the same keys will be used to authenticate the server and avoid the warning the clients might get if the host keys are modified. **Note**: saving key files in PEM format requires that the [Bouncy Castle](https://www.bouncycastle.org/) supporting artifacts be available in the code's classpath.
* HostKeyCertificateProvider - used for OpenSSH public-key certificate authentication system as defined in [this document](https://github.com/openssh/openssh-portable/blob/master/PROTOCOL.certkeys)
* ShellFactory - That's the part one usually has to write to customize the SSHD server. The shell factory will be used to create a new shell each time a user logs in and wants to run an interactive shell. SSHD provides a simple implementation that you can use if you want. This implementation will create a process and delegate everything to it, so it's mostly useful to launch the OS native shell. E.g.,

sshd.setShellFactory(new ProcessShellFactory(new String[] { "/bin/sh", "-i", "-l" }));

There is an out-of-the-box InteractiveProcessShellFactory that detects the O/S and spawns the relevant shell. Note that the ShellFactory is not required. If none is configured, any request for an interactive shell will be denied to clients.

Furthermore, one can select a specific factory based on the current session by using an AggregateShellFactory that wraps a group of ShellFactorySelector - each one tailored for a specific set of criteria. The simplest use-case is one the detects the client and provides a specially tailored shell for it - e.g., [the way we do for "WinSCP"](https://issues.apache.org/jira/browse/SSHD-1009) based on the peer client version string.

* CommandFactory - The CommandFactory provides the ability to run a **single** direct command at a time instead of an interactive session (it also uses a **different** channel type than shells). It can be used **in addition** to the ShellFactory.

SSHD provides a CommandFactory to support SCP that can be configured in the following way:

sshd.setCommandFactory(new ScpCommandFactory());

One can also use the ScpCommandFactory on top of one's own CommandFactory by placing the command factory as a **delegate** of the ScpCommandFactory. The ScpCommandFactory will intercept SCP commands and execute them by itself, while passing all other commands to the delegate CommandFactory

sshd.setCommandFactory(new ScpCommandFactory(myCommandFactory));

Note that using a CommandFactory is also **optional**. If none is configured, any direct command sent by clients will be rejected.

## Server side security setup

The SSHD server security layer has to be customized to suit your needs. This layer is pluggable and uses the following interfaces:

* PasswordAuthenticator for password based authentication - [RFC 4252 section 8](https://www.ietf.org/rfc/rfc4252.txt)
* PublickeyAuthenticator for key based authentication - [RFC 4252 section 7](https://www.ietf.org/rfc/rfc4252.txt)
* HostBasedAuthenticator for host based authentication - [RFC 4252 section 9](https://www.ietf.org/rfc/rfc4252.txt)
* KeyboardInteractiveAuthenticator for user interactive authentication - [RFC 4256](https://www.ietf.org/rfc/rfc4256.txt)

These custom classes can be configured on the SSHD server using the respective setter methods:

sshd.setPasswordAuthenticator(new MyPasswordAuthenticator());  
 sshd.setPublickeyAuthenticator(new MyPublickeyAuthenticator());  
 sshd.setKeyboardInteractiveAuthenticator(new MyKeyboardInteractiveAuthenticator());  
 ...etc...

Several useful implementations are available that can be used as-is or extended in order to provide some custom behavior. In any case, the default initializations are:

* DefaultAuthorizedKeysAuthenticator - uses the *authorized\_keys* file the same way as the SSH daemon does
* DefaultKeyboardInteractiveAuthenticator - for password-based or interactive authentication. **Note:** this authenticator requires a PasswordAuthenticator to be configured since it delegates some of the functionality to it.

## Configuring ciphers, macs, digest...

SSH supports pluggable factories to define various configuration parts such as ciphers, digests, key exchange, etc... The list of supported implementations can be changed to suit one's needs, or one can also implement one's own factories.

Configuring supported factories can be done with the following code:

sshd.setCipherFactories(Arrays.asList(BuiltinCiphers.aes256ctr, BuiltinCiphers.aes192ctr, BuiltinCiphers.aes128ctr));  
 sshd.setKeyExchangeFactories(Arrays.asList(new MyKex1(), new MyKex2(), BuiltinKeyExchange.A, ...etc...));

One can configure other security components using built-in factories the same way. It is important to remember though that the **order** of the factories is important as it affects the key exchange phase where the client and server decide what options to use out of each peer's reported preferences.

## Starting the Server

Once we have configured the server, one need only call sshd.start();. **Note**: once the server is started, all of the configurations (except the port) can still be *overridden* while the server is running (caveat emptor). In such cases, only **new** clients that connect to the server after the change will be affected - with the exception of the negotiation options (keys, macs, ciphers, etc...) which take effect the next time keys are re-exchanged, that can affect live sessions and not only new ones.

## Providing server-side heartbeat

The server can generate [SSH\_MSG\_IGNORE](https://tools.ietf.org/html/rfc4253#section-11.2) messages towards its client sessions in order to make sure that the client does not time out on waiting for traffic if no user generated data is available. By default, this feature is **disabled** - however it can be enabled by invoking the setSessionHeartbeat API either on the server (for **global** setting) or a specific session (for targeted control of the feature).

*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.

If one is using the SSHD CLI code, then these options are controlled via -o ServerAliveInterval=NNNN where the value is the requested **global** interval in **seconds**. *Note*: any non-positive value is treated as if the feature is disabled.

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.