# Commands infrastucture

## FileSystemFactory usage

This interface is used to provide "file"-related services/commands - e.g., SCP and SFTP - although it can be used for remote command execution as well (see the section about commands and the Aware interfaces). The default implementation is a NativeFileSystemFactory that simply exposes the [FileSystems.getDefault()](https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystems.html#getDefault) result. However, for "sandboxed" implementations one can use the VirtualFileSystemFactory. This implementation provides a way for deciding what is the logged-in user's file system view and then use a RootedFileSystemProvider in order to provide a "sandboxed" file system where the logged-in user can access only the files under the specified root and no others.

SshServer sshd = SshServer.setupDefaultServer();  
 sshd.setFileSystemFactory(new VirtualFileSystemFactory() {  
 @Override  
 public Path getUserHomeDir(SessionContext session) throws IOException {  
 ...use whatever information ...  
 return somePath;  
 }  
 });

The usage of a FileSystemFactory is not limited though to the server only - the ScpClient implementation also uses it in order to retrieve the *local* path for upload/download-ing files/folders. This means that the client side can also be tailored to present different views for different clients. A special "empty" NoneFileSystemFactory is provided in case no files are expected to be accessed by the server.

## ExecutorService-s

The framework requires from time to time spawning some threads in order to function correctly - e.g., commands, SFTP subsystem, port forwarding (among others) require such support. By default, the framework will allocate an [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) for each specific purpose and then shut it down when the module has completed its work - e.g., session was closed. Note that SSHD uses the CloseableExecutorService interface instead of the usual ExecutorService in order to provide graceful shutdown. Users may provide their own CloseableExecutorService(s) instead of the internally auto-allocated ones - e.g., in order to control the max. spawned threads, stack size, track threads, etc... but they can leverage the SshThreadPoolExecutor implementation which should cover most use cases.

Users who want to provide their own ExecutorService and not use SshThreadPoolExecutor should wrap it as a NoCloseExecutor and take care of shutting it down when SSHD is done with (provided, of course, that the user's own code does not need it to remain active afterwards...).

/\*  
 \* An example user-provided executor service for SFTP - there are other such locations.  
 \* By default, the SftpSubsystem implementation creates a single-threaded executor  
 \* for each session, uses it to spawn the SFTP command handler and shuts  
 \* it down when the command is destroyed  
 \*/  
 SftpSubsystemFactory factory = new SftpSubsystemFactory.Builder()  
 .withExecutorServiceProvider(() -> new NoCloseExecutor(mySuperDuperExecutorService))  
 .build();  
 SshServer sshd = SshServer.setupDefaultServer();  
 sshd.setSubsystemFactories(Collections.<NamedFactory<Command>>singletonList(factory));

If a single CloseableExecutorService is shared between several services, it needs to be wrapped with the ThreadUtils.noClose(executor) method.

CloseableExecutorService sharedService = ...obtain/create an instance...;  
  
 SftpSubsystemFactory factory = new SftpSubsystemFactory.Builder()  
 .withExecutorServiceProvider(() -> ThreadUtils.noClose(sharedService))  
 .build();  
  
 ChannelAgentForwarding forward = new ChannelAgentForwarding(ThreadUtils.noClose(sharedService));

**Note:** Do not share the instance returned by ThreadUtils.noClose between services as it interferes with the graceful closing mechanism. Use a new wrapper instance for each service.

## Remote command execution

All command execution - be it shell or single command - boils down to a Command instance being created, initialized and then started. In this context, it is **crucial** to notice that the command's start() method implementation **must spawn a new thread** - even for the simplest or most trivial command. Any attempt to communicate via the established session will most likely **fail** since the packets processing thread may be blocked by this call. **Note:** one might get away with executing some command in the context of the thread that called the start() method, but it is **extremely dangerous** and should not be attempted.

The command execution code can communicate with the peer client via the input/output/error streams that are provided as part of the command initialization process. Once the command is done, it should call the ExitCallback#onExit method to indicate that it has finished. The framework will then take care of propagating the exit code, closing the session and (eventually) destroy()-ing the command. **Note**: the command may not assume that it is done until its destroy() method is called - i.e., it should not release or null-ify any of its internal state even if onExit() was called.

Upon calling the onExit method the code sends an [SSH\_MSG\_CHANNEL\_EOF](https://tools.ietf.org/html/rfc4254#section-5.3) message, and the provided result status code is sent as an exit-status message as described in [RFC4254 - section 6.10](https://tools.ietf.org/html/rfc4254#section-6.10). The provided message is simply logged at DEBUG level.

// A simple command implementation example  
 class MyCommand implements Command, Runnable {  
 private InputStream in;  
 private OutputStream out, err;  
 private ExitCallback callback;  
  
 public MyCommand() {  
 super();  
 }  
  
 @Override  
 public void setInputStream(InputStream in) {  
 this.in = in;  
 }  
  
 @Override  
 public void setOutputStream(OutputStream out) {  
 this.out = out;  
 }  
  
 @Override  
 public void setErrorStream(OutputStream err) {  
 this.err = err;  
 }  
  
 @Override  
 public void setExitCallback(ExitCallback callback) {  
 this.callback = callback;  
 }  
  
 @Override  
 public void start(Environment env) throws IOException {  
 spawnHandlerThread(this);  
 }  
  
 @Override  
 public void run() {  
 while(true) {  
 try {  
 String cmd = readCommand(in);  
 if ("exit".equals(cmd)) {  
 break;  
 }  
  
 handleCommand(cmd, out);  
 } catch (Exception e) {  
 writeError(err, e);  
 callback.onExit(-1, e.getMessage());  
 return;  
 }  
  
 callback.onExit(0);  
 }  
  
 @Override  
 public void destroy() throws Exception {  
 ...release any allocated resources...  
 }  
 }

### Aware interfaces

Once created, the Command instance is checked to see if it implements one of the Aware interfaces that enables injecting some dynamic data before the command is start()-ed.

* SessionAware - Injects the Session instance through which the command request was received.
* ChannelSessionAware - Injects the ChannelSession instance through which the command request was received.
* FileSystemAware - Injects the result of consulting the FileSystemFactory as to the [FileSystem](https://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html) associated with this command.

### Data stream(s) sizing consideration

Some commands may send/receive large amounts of data over their STDIN/STDOUT/STDERR streams. Since (by default) the sending mechanism in SSHD is **asynchronous** it may cause *Out of memory* errors due to one side (client/server) generating SSH\_MSG\_CHANNEL\_DATA or SSH\_MSG\_CHANNEL\_EXTENDED\_DATA at a much higher rate than the other side can consume. This leads to a build-up of a packets backlog that eventually consumes all available memory (as described in [SSHD-754](https://issues.apache.org/jira/browse/SSHD-754) and [SSHD-768](https://issues.apache.org/jira/browse/SSHD-768)). As of version 1.7 one can register a ChannelStreamPacketWriterResolver at the client/server/session/channel level that can enable the user to replace the raw channel with some throttling mechanism that will be used for stream packets. Such an (experimental) example is the ThrottlingPacketWriter available in the sshd-contrib module. **Note:** if the ChannelStreamPacketWriterResolver returns a wrapper instance instead of a Channel then it will be **closed** automatically when the stream using it is closed.