Bitvise SSH Server

# Virtual Filesystem Provider Development Kit (SfsKit)

updated for SSH Server version 6.31

## Summary

A Bitvise SSH Server administrator can configure virtual filesystem layouts which a user can access by connecting to the SSH Server using SFTP or SCP. A mount point in the layout can map to a Windows filesystem directory served by the SSH Server, or to storage implemented by a pluggable virtual filesystem provider. The SfsKit provides tools, definitions, and an example to implement such a provider.

## Audience

SfsKit is intended for use by experienced C++ developers comfortable with creating project settings, building a Windows DLL, and interpreting header files and source code. For the most part, the code provided in SfsKit *is* the documentation. If this makes you uncomfortable, you may find SfsKit unsuitable.

## License

## Content

**SfsProvider**: Contains definitions of the interface a provider needs to implement in *SfsProvider.h*. Contains utility classes and methods a provider can (and should) use in *SfsUtilities.h/.cpp*.

**SfsWin**: Implements an example virtual filesystem provider that provides access to the Windows filesystem. Reproduces basic functionality of the default provider implemented by the SSH Server.

**Binaries**: Includes *SftpClient.exe* – a command-line SFTP test client exposing low-level access to SFTP requests, allowing for granular testing.

## Guidance

A virtual filesystem provider exposes its functionality via C-style functions *GetSfsVersion*, *HandleSfsInit*, *HandleSfsRequest*, and *HandleSfsDestroy*.

If a file transfer session uses the same virtual filesystem provider attached using different parameters to multiple mount points, this will be represented with multiple calls to *HandleSfsInit*. A provider must be able to handle this, and to handle *HandleSfsRequest* and *HandleSfsDestroy* calls appropriately, depending on the context passed to them.

See **SfsWin** for an example of functionality a virtual filesystem provider should implement. See **SfsProvider** for definitions of the interface that needs to be implemented.

All provider functions are called from fibers with a very **limited stack**. Refrain from using stack allocation, and do not use deep recursion.

Your implementation can use blocking I/O, but this can decrease performance and responsiveness of the file transfer session as a whole. To minimize impact on the rest of the file transfer session, use non-blocking I/O.

## Non-Blocking I/O

For non-blocking I/O, use *context->AddWaitObject*, *AddTimer*, and *Wait*. *Wait* may return even if set conditions are not met, so it must be called in a loop. **SfsWin** shows how to do so. Example:

```
while (true)
{
    if (!context->AddWaitObject(eventHandle) ||
        !context->AddTimer(context, timeout))
        throw ContextFailure();

    if (!context->Wait())
        throw ContextFailure();

    if (WaitForSingleObject(eventHandle, 0) != WAIT_TIMEOUT ||
        currentTime > timeoutTime) // pseudo-code
        break;
}
```

# Logging

In order for the SSH Server's file transfer logging, the Activity tab, and the On-upload command to work, the virtual filesystem provider must perform proper logging:

1. *SendLog* and *SendLogByHandle* must be called before *SendResponse*.
2. File transfer statistics must be reported using *TransferFile* (see *SfsUtilities.h*).
   **SfsWin** shows how to do so.
3. Within *HandleSfsDestroy*, closing of any still-open files must be logged using *SendLogByHandle*.
   **SfsWin** shows how to do so.
4. Within *HandleSfsRequest*, *SendLogByHandle* can be used instead of *SendLog* everywhere where *SFS_Request* contains a handle.

The *Log* class supports **generic** and **concrete** logging.

## Generic Logging

The following constructor is used for generic logging:

```
Log::Log(DefaultCode code, unsigned int reserve = 0);
```

In generic logging, *code* is either *Log::Success* (0), *Log::Failure* (200), or *Log::Progress* (600).

Generic logging is used to log an event with characteristics that are typical of, and related to, the SFTP action being executed. For example, while handling *SFS_RequestType::Open*, construct a *Log* object generically using *Log(Log::Failure)* to indicate that the open request failed.

## Concrete Logging

The following constructor is used for concrete logging:

```
Log::Log(unsigned int code, wchar_t const* desc, bool copyDesc,
         unsigned int reserve = 0);
```

In concrete logging, *code* is constructed as *SFS_LogAction* + *SFS_LogStatus. SFS_LogAction* is a multiple of 1000 corresponding to an SFTP action, defined in *SfsProvider.h.* Allowable values for *SFS_LogStatus* range from 0 – 199 for *Success*; 200 – 599 for *Failure*; and 600 – 999 for *Progress*.

Concrete logging is used to report actions taken in addition to the action being executed. For example, when executing *HandleSfsDestroy*, the implied action is *SFS_LogAction::Destroy*. In order to report closing of still-open files, use concrete logging to report *SFS_LogAction::Close*. **SfsWin** shows how to do so.

You can also use concrete logging if you wish to record your own *SFS_LogStatus* code, along with your own description. For an example, see *SfsWin.cpp*:

```
enum { SetFileTimeFailureAsProgress = 601 };
wchar_t const* message =
    L"Setting file time for the newly created directory failed.";
Log(SFS_LogAction::MkDir + SetFileTimeFailureAsProgress, message, ...
```

Every *SFS_LogStatus* code is intended to have its own corresponding description, and vice versa.

# References

The SFTP implementation in Bitvise SSH Server and Client, as well as the structure of the SSH Server's virtual filesystem, is based on SFTP protocol version 6, defined in the following documents:

https://tools.ietf.org/html/draft-ietf-secsh-filexfer-13

https://tools.ietf.org/html/draft-galb-filexfer-extensions-00

We recommend keeping these specifications handy during development.